1994-06

Remote sensing, processing and transmission of data for an unmanned aerial vehicle

Howard, Donald Benton.

Monterey, California. Naval Postgraduate School

Approved for public release; distribution is unlimited.

Remote Sensing, Processing and Transmission of Data for an Unmanned Aerial Vehicle

by

Donald Benton Howard
Lieutenant, United States Navy
B.S., University of Kansas, 1988

Submitted in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
June 1994

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	June 1994	Master's Thesis

4. TITLE AND SUBTITLE REMOTE SENSING, PROCESSING AND TRANSMISSION OF DATA FOR AN UNMANNED AERIAL VEHICLE (U)	5. FUNDING NUMBERS
6. AUTHOR(S) Howard, Donald Benton	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey CA 93943-5000	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION/AVAILABILITY STATEMENT	12b. DISTRIBUTION CODE
Approved for public release; distribution is unlimited.	A

13. ABSTRACT *(maximum 200 words)*

This thesis chronicles the development of a proof-of-concept, stand-alone, Unattended Ground Sensor (UGS) that can be used to sense and process signals associated with the motion of large vehicles, troops, or aircraft. The results of this signal processing are then transmitted to an Unmanned Aerial Vehicle (UAV). The UGS uses acoustic and seismic sensors to provide data to a Digital Signal Processing (DSP) computer. Digital signal processing algorithms can be independently developed in the C programming language and linked with the software developed for this project.

14. SUBJECT TERMS Unmanned Ground Sensor, UAV, Digital Signal Processing			15. NUMBER OF PAGES 98
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

ABSTRACT

This thesis chronicles the development of a proof-of-concept, stand-alone, Unattended Ground Sensor (UGS) that can be used to sense and process signals associated with the motion of large vehicles, troops, or aircraft. The results of this signal processing are then transmitted to an Unmanned Aerial Vehicle (UAV). The UGS uses acoustic and seismic sensors to provide data to a Digital Signal Processing (DSP) computer. Digital signal processing algorithms can be independently developed in the C programming language and linked with the software developed for this project.

TABLE OF CONTENTS

ACKNOWLEDGMENT

I would like to thank Professor Shields and Professor Tummala for their patience and guidance during the development of this thesis. Their enthusiasm for research is very contagious. I would also like to thank Gordon Sterling of Analog Devices and Farid Dibachi of Wavetron Microsystems for enduring my many questions. Most importantly, I would like to thank Alice, Caitlin, and Christianna for their support and understanding.

I. INTRODUCTION

A. PROBLEM STATEMENT

Unmanned Aerial Vehicles (UAV) have been under development for some time. They have enjoyed a degree of success in operational environments. Recently, during Operation Desert Storm, it became evident that UAVs could make a significant contribution to a warfighting effort. As technology has advanced, so too has the number of possible missions that can be undertaken by an UAV. Much of the progress in UAV technology can be attributed to the development of fast, small computers. The Naval Postgraduate School is taking advantage of these advances in computer technology and electronic miniaturization in an ongoing research project to develop several UAV platforms including a Vertical Takeoff and Landing (VTOL) UAV called Archytas. As this project nears the operational testing phase, more and more missions are being envisioned.

One of the missions being considered is the gathering of intelligence from ground detectors. The detectors could also take advantage of state-of-the-art miniaturization and computer technology. Many small low power digital signal processing (DSP) circuit boards are presently on the market.

These DSP boards could be used to sample data from sensors, process this data, and send only the results to an UAV.

B. THESIS SCOPE

This thesis describes the design and construction of an inexpensive, compact, self-contained detection device, capable of processing and transmitting data. The device was designed using current off-the-shelf (COTS) technology hardware. The process used to design, build, and test such a device is detailed in the following chapters. This device will be referred to as Unattended Ground Sensor (UGS).

Chapter II is a background chapter describing the suitability of an UAV as an intelligence gathering device. There is also a discussion in this chapter about the history and the future of UGS systems. Chapter III addresses the hardware design requirements for the UGS. The fourth chapter discusses possible solutions to the design problem. It also describes the equipment chosen to fulfill the design requirements. Chapter V covers the software design objectives. It also describes the programming philosophy and solutions used in this design. The concluding remarks and recommendations for future development are contained in Chapter VI.

The device described herein is not suitable as the final solution to the UGS/UAV combination. It is a technology demonstrator and a good test bed for DSP algorithm testing.

It is also an excellent platform for UAV/UGS data link testing.

II. BACKGROUND

The timely collection and evaluation of battlefield intelligence has always been paramount to the success of any warfighting scenario. This is still very true in today's world. Recent advances in computer and micro-controller technologies have breathed new life into some old techniques of intelligence gathering.

The Naval Postgraduate School (NPS) in Monterey, CA has been involved in a project that makes use of these advances in technology to design Unmanned Aerial Vehicles (UAVs). The potential of the UAV is best stated in the DoD, 1993 Unmanned Aerial Vehicles (UAV) Master Plan:

> Unmanned Aerial Vehicles (UAVs) can make significant contributions to the warfighting capability of operational forces. They greatly improve the quality and timeliness of battlefield information while reducing the risk of capture or loss of troops, thus allowing more rapid and better informed decision making by battlefield commanders. They are cost effective and versatile systems. While reconnaissance, surveillance, and target acquisition (RSTA) are the premier missions of UAVS, they can also provide substantial capabilities in electronic warfare (EW), electronic support measures (ESM), mine detection, command and control and special operations mission areas. UAVs are a particularly valuable adjunct to the Services' aviation communities. They can readily perform a multitude of inherently hazardous missions: those in contaminated environments, those with extremely long flight times and those with unacceptable political risks for manned aircraft. Allotting these dirty and dangerous missions to UAVs increases the survivability of manned aircraft and frees pilots to do missions that require the flexibility of the manned system. UAVs are a viable alternative as the Services wrestle with the many challenges of downsizing the force structure.[Ref. 1]

As indicated above the potential tasks for an UAV are nearly limitless.

One enhancement to the RSTA mission mentioned above is to use the UAV as a central collection point for intelligence transmitted by Unattended Ground Sensors (UGS). With current technology, it is feasible to build a small UGS that can sense environmental parameters, detect changes in them, and identify the source that caused these changes. All of this can be done by a small standalone unit the size of a 12 oz. soda can.

This chapter will discuss the characteristics unique to an UAV that make it ideally suited for intelligence gathering. In addition, it will describe the employment and contributions of unmanned ground sensors in the Vietnam Conflict. Finally, this chapter will discuss the potential of devices similar to those used in Vietnam that utilize current technology.

A. UNMANNED AERIAL VEHICLE (UAV)

Currently there is considerable interest in the research and development of Unmanned Aerial Vehicles (UAVs). UAVs are an excellent compromise between the inexpensive, dumb drone and the expensive, but versatile manned aircraft. This section describes the UAV's suitability for intelligence gathering.

1. The UAV's Suitability for Intelligence Gathering

Ideally an intelligence gathering device would have the following qualities:

- Inexpensive.
- Expendable.
- Low probability of being detected by the enemy.
- Low probability of destruction by enemy.
- Low failure rate.
- Mission flexible.
- Minimum risk to friendly lives.

A drone is relatively inexpensive when compared to a manned aircraft and is often expendable. However, it lacks mission flexibility once it has been launched. For example, if the drone detected an event worth further investigation during its flight, it could not turn around and take a second look. This inability to react to real time occurrences is a severe limitation in intelligence gathering. On the other hand, a manned aircraft is infinitely more flexible, but it also puts human life at risk. This means that the situations most worth investigating will not be pursued because they are too risky to human life.

Recent advances in technology have made it feasible to build a highly survivable, High-Altitude-Long-Endurance (HALE) UAV. The issue of survivability can be broken down into two separate categories, susceptibility and vulnerability. Susceptibility is a measure of the ease with which a system can be detected and attacked. Vulnerability is an indication of a systems ability to continue its mission once it has been

attacked. Both susceptibility and vulnerability must be minimized to increase survivability. One method to reduce susceptibility is to give the system the ability to defend itself. Susceptibility can also be minimized by reducing radiated or reflected signatures. The second method decreases susceptibility by lowering the probability of detection and thereby, maintaining covertness. If the UAV is not detected, it will not be fired upon.[Ref. 2]

The physical characteristics of an UAV will inherently minimize its probability of being detected. The advent of miniaturized electronics allows for the use of small, powerful computers for flight control. Other crucial electronic systems, such as the Inertial Navigation System (INS) and the Global Positioning Systems (GPS), have also been reduced in size. These developments tend to reduce the airframe size. It is obvious that a small UAV is harder to be seen than a larger manned aircraft. This is particularly true when the small UAV is flying at a high altitude. However, size can also be related to the radar cross-section of the UAV. Since UAVs are designed to carry smaller payloads than manned aircraft, their airframes are smaller. This can translate into a smaller engine. The use of nonmetallic, composite materials in airframe construction also tends to reduce the total weight and by that allow for a smaller engine. By using smaller engines (less metal) and nonmetallic materials in the

airframe construction, the radar cross-section can be reduced to achieve a lower probability of detection.[Ref. 2]

The use of a small engine in the UAV has the potential to reduce the amount of radiated noise and heat to a level below that generated by a manned aircraft. The minimization of heat and noise generation will also lower the probability of UAV detection.[Ref. 2]

To be an effective real-time intelligence gathering tool, the UAV must transmit data to a control station. The process of transmitting could compromise the UAV and make it more detectable. Fortunately, the use of Spread Spectrum communication techniques can reduce the probability of detection to acceptable levels.[Ref. 2]

Vulnerability is the other component of survivability. Vulnerability is a measure of the inability of a system to continue operating after it has taken a "hit." One method to lower the vulnerability of a system is to make it tougher. This can be done by physically protecting vital systems. Tough composite materials such as Kevlar or Glass Reinforced Plastic (GRP) used in the construction of many UAVs will act like Armor protecting vital systems and reducing vulnerability.[Ref. 2]

Cost as well as survivability must be considered in the evaluation of an intelligence gathering system. The cost of an UAV is usually between that of a drone and a manned aircraft. Normally the cost of an UAV is too high to consider

it expendable, but low enough to make it a cost-effective alternative to a manned aircraft.

An UAV can be an inexpensive, highly-survivable, mission-flexible tool. Additionally, because UAVs have a much lower probability of detection than manned aircraft, they are less likely to effect the actions of the enemy and thus provide better intelligence. All of these factors combine to make a HALE UAV the perfect instrument for gathering intelligence.

B. UNATTENDED GROUND SENSOR (UGS)

The Unattended Ground Sensor can play an integral part in battlefield intelligence gathering systems. Their utility was proven during the Vietnam conflict.[Ref. 3] Even greater use can be made of the UGS by incorporating some of today's technology.

1. Uses in the Vietnam Conflict

Vietnam saw the first use of Unattended Ground Sensors using a radio link to a remote monitor [Ref. 4]. In Reference 5, John Bergin states:

> In 1966 the United States began to build an electronic barrier of acoustic, seismic, and radio sensors across the northern border of South Vietnam, the panhandle of Laos, and the eastern regions of Thailand to detect North Vietnamese infiltration.[Ref. 5]

Bergin goes on to explain that these sensors were seeded by air along the McNamara line. They relayed the detected sounds to on-station monitoring aircraft. This information was then

used to vector in air strikes against the most promising targets. Around 1968 General Westmoreland started using some sensors to monitor the perimeters of Marine bases. "They were so successful in warning of enemy movements and identifying targets for artillery and air support that General Westmoreland obtained permission to postpone the completion of the McNamara Line to use the sensors in tactical operations."[Ref. 5]

The reviews of these new state-of-the-art devices were nearly all-positive. Assertions were made at Congressional hearings on the electronic battlefield that these devices were responsible for lowering the American death rate from 12 percent to 3 percent. In addition, claims were made that a battalion with the sensors could monitor twice the area of a nonsensor equipped battalion.[Ref. 3]

Although these sensors were successful, they had some serious drawbacks. The first drawback was that they were bulky thus making them difficult to deploy and easy to sight. This problem was partially addressed by camouflaging them as vegetation.[Ref. 5]

The second drawback was due to the short battery life of the detectors. When the sensor batteries were exhausted or near the end of their useful life, patrols had to be deployed to replace the batteries. These battery replacement patrols exposed the troops to unnecessary dangers.

A third drawback to the Vietnam era UGS was their inability to process raw data. Reference 4 states, "The sensor signal must be processed and transmitted over a data link, should ideally only transmit when a significant event has occurred ...." The Unmanned Ground Sensors used in Vietnam had to continuously transmit their data to a nearby aircraft. This made the sensors vulnerable to detection and location by radio direction finding equipment.

The final drawback stemmed from the fact that manned aircraft were used to investigate sensor events. Unlike an UAV, these aircraft were large and loud making them counter detectable by the simplest of means. This allowed the enemy the opportunity to alter their actions and avoid further detection.

Since the Vietnam war, the UGS concept has evolved through a series of programs. The Army started a program in 1972 to develop an all-weather, all-terrain, REmote Monitored Sensor System (REMBASS). This system was intended for division level use and incorporated the Platoon-level Early Warning System (PEWS). The PEWS became operational in 1980. An Improved REMBASS (IREMBASS) system began development in the late 1980's. The system developed here will be very much like REMBASS, but with improved signal processing capabilities.[Ref. 6]

2. Uses in Tomorrow's Conflicts

All of the applications above have one thing in common. They have relied on either a manned aircraft or ground station to collect and relay or analyze the data from the Unattended Ground Sensors. It is possible, with the advances in computing power and data storage, to build an UGS that can analyze the sensed data on its own. The UGS would then only need to transmit to relay stations or an UAV when the detection criterion has been met. If this UGS technology were to be coupled with an UAV as the receiver, sensor detections could be immediately investigated with a low probability of counter detection. The UAV could then transmit video and infrared information to a command center for further evaluation.

C. SUMMARY

Although sensor systems in the past have been successful, their performance could be enhanced further with the application of modern technology. The UGS/UAV combination has great potential for safe, inexpensive, accurate, and stealthy intelligence gathering. The application of this system could range from the detection of men and vehicles to the detection of a ballistic missile launch.

III. UGS HARDWARE DESIGN CONSIDERATIONS

As is true for any system, many design factors must be considered to produce the best product possible. Among these factors are capabilities, cost, size, weight, and power requirements. This proof-of-concept UGS must meet the following general design requirements. It should be a highly capable, standalone, digital signal processing unit that is small enough to fit in a container the size of a 12 oz. soda can. It must also be capable of sensing various environmental parameters, processing this information, determining if the detection criterion is satisfied, and notifying the UAV that a detection has occurred.

The primary goal of this design is to produce a product that meets all of the general design requirements. In addition, it must be constructed using low cost, commercial off-the-shelf (COTS) components. This chapter details the design approaches considered in each of the four major hardware categories.

A. SENSING DEVICE REQUIREMENTS

There were five environmental parameters that were considered in developing this device. These parameters were chosen for evaluation based on their possible applicability and on the environmental sensors that are readily available on

the commercial market. The sensors considered for use in this design were acoustic, seismic, pressure, magnetic, and global position sensing (GPS) devices. This section discusses each of these sensors and their applicability to this project.

1. Acoustic Sensor

An acoustic sensor can be a simple microphone. This type of sensor has several advantages. Microphones are small, light weight, and have very low energy consumption. They are also readily available. As a sensor it can be used to detect an object of interest when that object is loud, but not in close physical proximity. Most microphones have a cycloid beam pattern. This can be a disadvantage if the orientation of the microphone cannot be controlled or if the exact location of the target of interest is not known. There are some microphones available, however, that have good fidelity ranges and a hemispherical beam pattern. It was determined that a microphone with a hemispherical beam pattern and a frequency response of 50 Hz to 18,000 Hz was necessary for our particular application.

2. Seismic Sensor

A seismic sensor can measure very small movements of the sensor itself. If the sensor is in solid contact with a surface, it will measure the movement of that surface. These movements can be detected at frequencies as low as 10 Hz.

Seismic sensors can vary greatly in size, weight, and cost. One sensor investigated weighed 3.3 pounds and had a volume of 25.8 in^3. Another sensor weighed only 1.8 oz and had a volume of 0.61 in^3. The cost of these sensors ranged from as high as several thousand dollars to as low as a few hundred dollars.

The design requirements specify that the UGS must be small; therefore, the seismic sensor should be as small as possible. In addition, it was determined that the UGS should be sensitive to seismic signals within the frequency range of 1 to 1000 Hz.

3. Pressure Sensor

Pressure sensors are most useful in applications where the pressure transmitting medium is dense or the sensor is located very near the pressure source. It was determined that neither of these situations would apply for our specific application. As a result, no pressure sensor is included in the UGS design at this time.

4. Magnetic Sensor

The magnetic sensor can detect disturbances in the local magnetic field. These disturbances could be caused by the passing of an iron based metal object or a current carrying conductor in the proximity of the sensor. A tank, for example, would have to pass within approximately one tank length of the magnetic sensor to be detected.[Ref. 7] It is

anticipated that the UGS will not be close enough to the object of interest to make use of a magnetic sensor. Consequently, a magnetic sensor is not included in the current UGS design.

5. Global Position Sensor (GPS)

GPS, if used, could determine the position of the UGS. This information could be included in any communication with the UAV. The position data could then be used by the UAV to pinpoint the detection location or track a disturbed UGS. However, for this application it assumed that the UGS's position will be known at all times. The use of a GPS sensor is, therefore, not required.

B. TRANSMISSION REQUIREMENTS

The UGS must be able to periodically transmit an "I am OK" signal and aperiodically transmit an "Alarm" signal if the processed data indicates that a detection has occurred.

1. Message Format

The transmitted message must indicate the identity of the UGS and whether an alarm condition exists. In addition, it should encode the data to provide some error detection and correction capability.

There are two different categories of information coding:

- *Convolutional codes* continuously encode long bit streams. This method is best suited for continuous transmissions.

- *Block codes* group the data into blocks and encode these blocks as a unit.

This system will only be transmitting blocks of data; therefore, convolutional coding is not necessary. Use of block coding is appropriate here.[Ref. 8]

Several error detection/correction methods use block coding. One such method adds a single parity bit to the data block. This method will detect a single error, but will not provide any error correction capability. Hamming Code is another method, which uses 2^N-1 bits to describe 2^N-1-N data bits. It can be used to detect and correct only single bit errors: single error detection, single error correction (SEDSEC). Adding another bit to the Hamming scheme above to indicate overall parity will allow for the detection of two bit errors. This scheme is called double error detection, single error correction (DEDSEC). It requires 2^N bits to encode the message and provides for 2^N-1-N data bits.

2. Transmitter Interface

The transmitter must be able to communicate with the digital signal processor (DSP) board. This can be done via an RS-232 serial communication port. Some DSP boards provide these ports, while others do not. The UAV project currently being developed at the Naval Postgraduate School, Monterey, CA is using an intelligent RS-232 controller with an available channel for communication with the UGS.

Another method of communication between the processor and radio is to use the available output ports of the DSP board. The types of output on the available DSP boards vary. Most of the DSP boards surveyed provide an analog output. However, one board investigated also provides a software controlled serial output. Either the TTL compatible output or the analog output could be used to communicate with a radio.

3. Transmitter Requirements

Besides satisfying the interface requirements, the transmitter should have low energy consumption, be battery powered, small, light weight, and secure. It also must be able to transmit to the UAV at a slant distance up to 30 Km. A link budget was calculated to determine the transmitter power required for this data link. Below is a list of the assumptions made in calculating the link budget:

1. Transmit from ground to air.
2. Maximum slant range: R = 30 Km.
3. Transmission antenna: Monopole with 2.3 dB peak gain.
4. Pointing loss for transmission antenna: 5 dB
5. Transmit frequency: 900 MHz (spread spectrum)
6. Receive antenna: Double skirted ground plane with 6 dB gain.
7. Pointing loss of receiving antenna: 1.0 dB
8. Moderate weather conditions.

As can be seen in Table I below the link budget calculations indicate the transmitter must be able to radiate 1.55 watts of power for adequate communication.

Table I UGS/UAV LINK BUDGET

Transmit Tx	Plus	Minus
Tx Antenna Gain (peak)	2.3 dB	
Tx pointing loss		5.0 dB
Propagation		
Free Space loss$(\lambda/4\pi R)^2$		121.2 dB
Atmospheric absorption		0.0 dB
Precip absorption		0.0 dB
Receive Rx		
Rx Antenna gain	6 dB	
Rx Pointing		1.0 dB
Noise Power		
Rx Noise (sensitivity=1 μv)		-107.0 dBm
Totals	8.3 dBm	20.2 dBm

Difference = -11.9 dbm

For a +20 dB signal margin, TX power Pt must be 31.9 dBm or 1.55 watts.

C. DIGITAL SIGNAL PROCESSOR (DSP) REQUIREMENTS

There are many DSP boards being marketed. These boards vary in speed, cost, I/O capabilities, signal processing capabilities, size, energy consumption, developmental support, and technical support. The discussion below describes each of these variables and their importance to this design.

1. Speed

There are two aspects of speed to consider for this design. The first is processor speed. The processor must be able to analyze the sampled data in a timely manner. This is necessary to facilitate the transmission of an alarm signal within a few seconds of a detection. Also, the faster the processor, the more complicated an analysis can be carried out.

The second aspect of DSP board speed is the rate at which a signal can be sampled. The board must be capable of sampling the input ports at or above the Nyquist rate, a rate equal to or greater than twice the highest frequency present in the signal applied to the ports.

2. Cost

The DSP board is the most expensive component of this design. The cost will be kept low by using COTS DSP boards. In addition, the cost can be kept low by purchasing only what is needed to fulfill the design requirements. Some allowance

can be made for expanded capabilities if the cost remains reasonable.

3. I/O Capabilities

The DSP board must be able to accept at least two analog input signals and have one output signal to communicate with a transmitter. The output data can be transferred via an analog port, a serial port, or a parallel port.

4. Signal Processing Capabilities

Digital signal processing is heavily based on arithmetic operations. The DSP board should use a microprocessor that easily executes arithmetic instructions such as multiplication, division, and shift operations.

The processor should also be able to perform bit reversal to aid in the calculation of Fast Fourier Transforms (FFT's). In addition it should have onboard memory for storing raw and processed data.

5. Size

The UGS is to fit in a volume approximately the size of a soda can. Consequently, the DSP board must be as small as possible. Most DSP boards investigated required less than 25 in^3.

6. Energy Consumption

The DSP board is the single largest load of all UGS components. To keep the total energy consumption low, the power required by the DSP board must be kept to a minimum.

This can be achieved by choosing a DSP board that makes use of low power CMOS technology. Additionally, some processors also have a sleep mode permitting them to conserve energy.

7. Developmental Support

One of the biggest considerations in choosing a DSP board is the developmental support that is provided. The DSP system purchased should provide for the development, debugging, and testing of user written code. This is vital in the early stages of product development. However, it is not as important once the user has a satisfactory algorithm tested and loaded on the DSP board.

8. Technical Support

Most current DSP systems are relatively new, this means the accompanying documentation may be incomplete or in error. It should be established that the technical support department of the DSP board manufacturer is cooperative and eager to help resolve any inaccuracies in the documentation or technical problems.

D. POWER SUPPLY REQUIREMENTS

The UGS is to operate in a stand-alone mode. This requires that the DSP board, microphone, and transmitter are powered by batteries. The size limitation of the entire unit dictates that the battery cannot be too large. This unit, however, must continue to operate for prolonged periods of time. Both requirements put severe limitations on the energy

consumption of the DSP board since it is the largest load on the power supply.

Most currently available DSP boards operate at five volts and require 2-3 watts of power. For a three watt device this means there is less than 11 hours of operational life for a 12 volt, 6.5 Amp-hr lead acid battery regulated to five volts.

To increase the operational lifetime of the system, the following alternatives should be investigated:

- Reduce system loading by replacing the DSP board with a low power version.
- Use efficient, high capacity batteries.
- Supplement the battery with solar power.

This proof-of-concept design will not explore these alternatives. Instead, this system will be developed using both non-rechargeable batteries and sealed, lead-acid batteries. It is anticipated that a production system would use CMOS ASIC technology, reducing power consumption by a factor of 1000.

E. SUMMARY

The design requirements will be met if the UGS is a stand-alone unit that uses acoustic and seismic sensors to detect and evaluate objects of interest. The evaluation will be performed by a digital signal processing board capable of receiving at least two input signals. In addition, the DSP

board must be able to send information packets indicating an "alarm" or "I am OK" condition to a transmitter. The transmitter will be a spread spectrum UHF system that receives these packets of information and relays them to the UAV. All power for the UGS will be provided by batteries.

IV. UNATTENDED GROUND SENSOR HARDWARE

Previous chapters provide a brief background of the UGS project and its design considerations. This chapter describes the components that are considered for incorporation in this project. It will also describe, in detail, the components that are included in the final product and the reasons for choosing them. Figure 1 shows a block diagram of the UGS system as implemented.

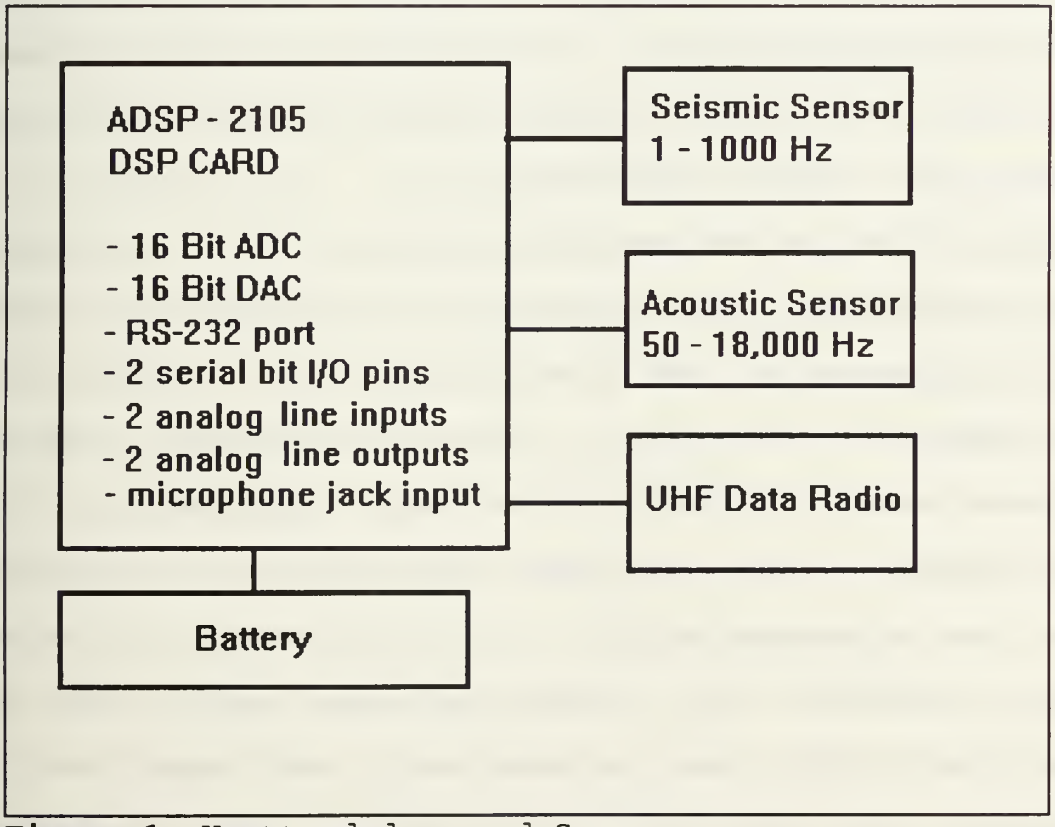


Figure 1 Unattended ground Sensor

A. DIGITAL SIGNAL PROCESSOR BOARD

The DSP board is the most important component of the UGS system. As such, the proper selection of a DSP board was paramount to the success of this design. Some design requirements of Chapter III were given more weight than others. The principal factors considered in this design were signal processing and I/O capabilities. Since this project was unfunded, the cost was another important factor. The next level of priority was assigned to the amount of developmental and technical support provided by the manufacturers. Finally, the lowest priority design considerations were processor speed and energy consumption. It was felt that, since this was a proof-of-concept device, faster and more energy conservative alternatives could be explored later.

Three DSP systems were considered for use in the UGS. A table summarizing these systems and their attributes can be found in Appendix A. The first of these was the Piranha 3111. This DSP board is based on a Texas Instruments TMS320C31 processor made by DSP Research and costs approximately $1500. The Piranha 3111 runs at 40 MHz, consumes about two watts of power, and communicates with a motherboard via a full duplex serial interface. The Piranha Evaluation Board (PEB) is a developmental board that can hold one or two Piranha modules. By using the PEB the Piranha modules can communicate individually with a host PC. They can also communicate with each other in a stand-alone mode by supplying the PEB with

+5/12 volts. The main drawback of the Piranha 3111 is that it has only one analog I/O channel. The Piranha was not chosen for development because of its limited technical support and limited I/O capabilities.

The second DSP board considered was the SBC-31 manufactured by Innovative Integration of Moorpark, California. The SBC-31 is a stand-alone processor also based on the Texas Instruments TMS320C31, 32-bit floating-point DSP. The board is very versatile with regard to I/O capabilities. The SBC-31 supports two fully duplexed RS-232C serial channels and a 48-bit digital I/O port. The fully configured SBC-31 supports analog inputs and outputs. The analog input channels are sampled at 200 kHz and have 16 bit resolution. They can be configured for 16 single-ended inputs, or eight differential inputs; or four differential and eight single ended inputs. The SBC-31 has four 16-bit, 200 kHz analog output channels. When running at 50 MHz, it is capable of 25 MIPS sustained performance.[Ref. 9] This board was delivered with its development package for $2500.00 and very good technical documentation. The cost of the SBC-31 without the development package is $1195.00. The biggest drawback for this board is its power consumption. The fully configured SBC-31 consumes about three watts of power. This high power consumption is due, in large part, to its high signal sampling rate and A/D conversions. The SBC-31 may be too capable for this project. A slower, less capable board might consume less

power and still meet the basic design requirements. However, further investigation of an UGS system based on the SBC-31 could prove fruitful. Development of a second UGS system based on the SBC-31 is currently planned by NPS.

The third and final DSP board considered was the EMB-1601A manufactured by Wavetron Microsystems of Redwood City, California. The EMB-1601A, a block diagram of which is shown in Figure 2, is a digital signal processing board based on Analog Devices' ADSP2105 chip. This DSP board has two 16-bit analog line inputs with RCA connectors, one microphone input, and two 16-bit analog line outputs with RCA connectors. It can sample the inputs at user determined frequencies that range from 5.5125 kHz to 48.00 kHz. Additionally, the EMB-1601A can be configured with a ten Mb/s synchronous serial port and/or a 9600 bps, RS-232, serial port.[Ref. 10] The RS-232 port with an optional development package permits communication with a PC. This combination allows the user to develop code on a PC and then download that code onto the EMB-1601A for testing. Besides the I/O ports mentioned above, the EMB-1601A has two parallel I/O pins that can be used for signaling or processor control. The EMB-1601A has a 10MHz crystal that allows it to execute instructions at a rate of

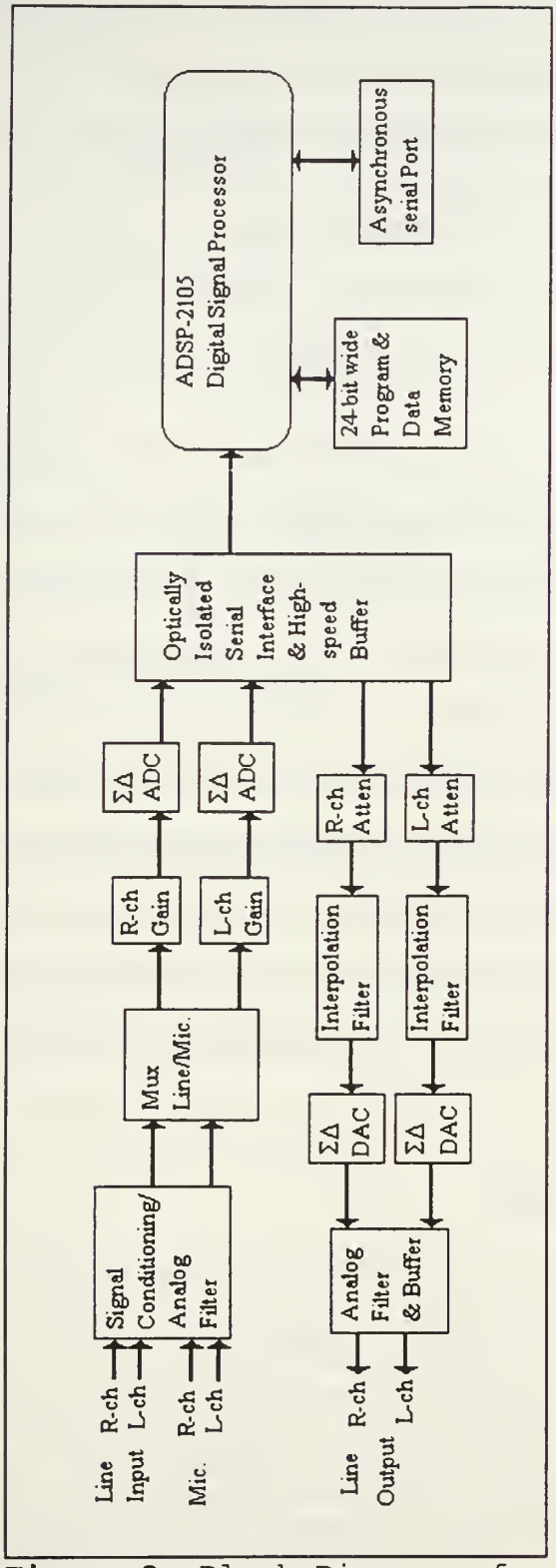


Figure 2 Block Diagram of EMB-1601A from [Ref. 10]

10 MIPS.[Ref. 11] It can operate in a stand-alone mode with a power consumption of about two watts.

This unit was shipped to NPS with the following options:

- 8K x 24 zero wait state SRAM.
- RS-232 based development software and C command library.
- DSP Assembler, Linker, and Simulator.

The total cost of this package was $1780.00. Some of the cost was due to the onetime purchase of the development packages. The cost of just the DSP board with SRAM was only $590.00.

There were several errors detected in the technical documentation provided by Wavetron Microsystems and Analog Devices. However, representatives from both companies were very helpful and all noted discrepancies were quickly resolved either by phone conversation or electronic mail. The EMB-1601A meets all of the design requirements and is suitable for use in this UGS system. The board layout for the EMB-1601A is shown in Figure 3.

B. SENSING DEVICES

There were two environmental properties considered best suited for exploitation in this UGS. This project will make use of airborne sound and low frequency ground vibrations by sensing acoustic and seismic information. A microphone will be used to sense the acoustic information. The specific

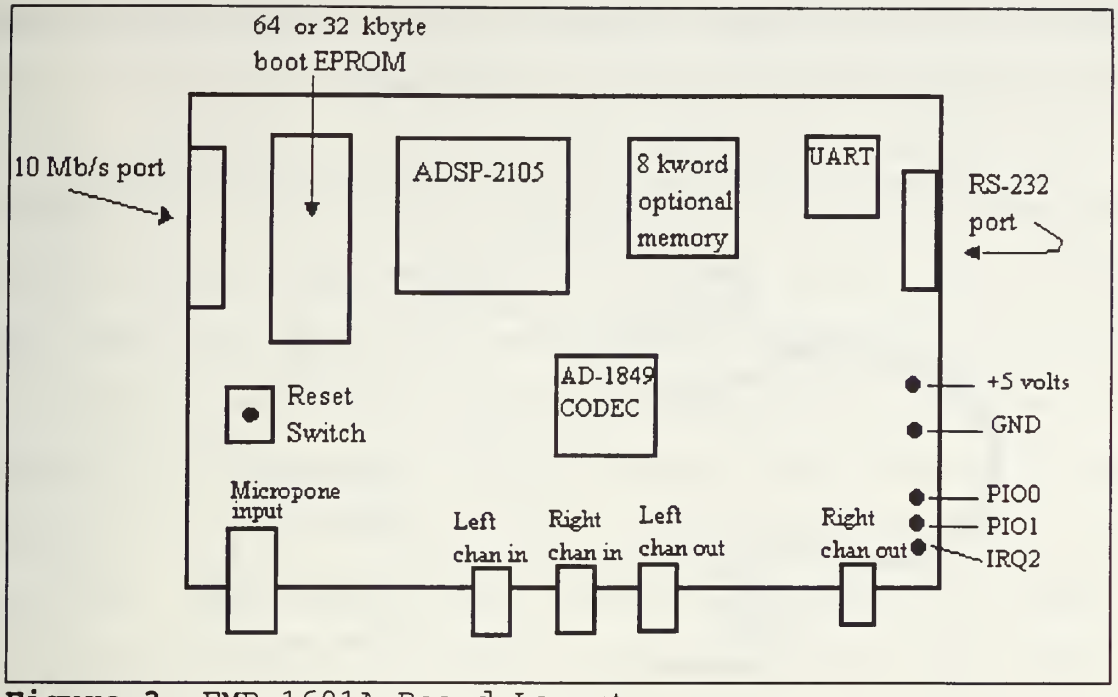


Figure 3 EMB-1601A Board Layout

microphone that was chosen is a Pressure Zone Microphone PZM-180 made by Crown of Elkhart, Indiana. This microphone has a hemispherical polar pattern that allows it to pick up sounds around it at sound pressure levels as high as 120 dB. The PZM-180 has a frequency response from 50 Hz to 18,000 Hz, with an open-circuit sensitivity of 3.2 mV/Pa. This microphone costs $190.00, and its characteristics make it ideal for the UGS project.

The seismic information will be sensed by an Oyo Geospace HS-J-K3A geophone. This sensor was chosen for its small size and weight (0.6 in^3 volume and 1.8 oz. weight). Two of these sensors along with their calibration data and mounting devices were donated to this project by the Naval Surface Weapons

Center, Silver Springs, Maryland. Figure 4 shows the geophone as wired for connection to the DSP board.

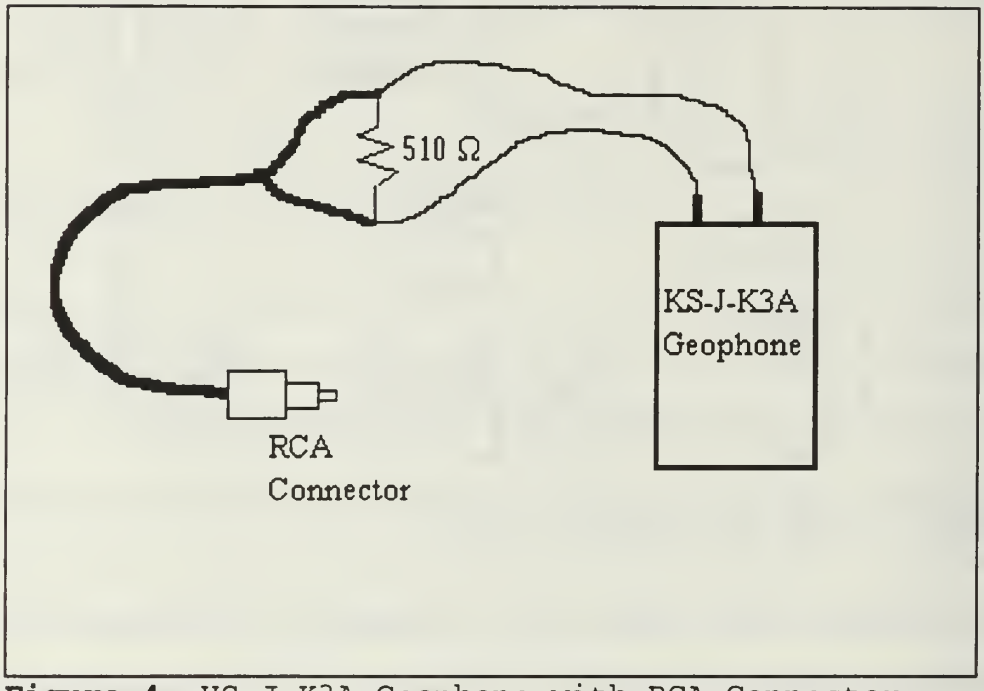


Figure 4 HS-J-K3A Geophone with RCA Connector.

C. TRANSMITTER

Since the RS-232 port of the DSP board was needed for development of this system, it was decided to use the two parallel I/O (PIO) pins of the DSP board to send signals to the transmitter. The PIO pins are TTL compatible and software controlled. The software for this project was developed based on the assumption that the transmitter chosen would take one TTL input to enable transmission and another TTL input that would carry the actual signal. Such a transmitter has not yet been identified. However, an interim solution has been found

for testing purposes only and can be used until a transmitter with the desired properties is identified or built.

A Lack of funding has prevented purchasing a transmitter/receiver pair. Nonetheless, the device identified as the temporary solution for use in this project is a 27.145 MHz, phase modulated, four watt, battery powered transmitter. It is made by Linear of Carlsbad, California and comes in three different models (MR161T, MR164T and MR168T). The models are differentiated based on the number of inputs they can take. For example, model MR164T can encode and transmit a unique message to the receiver for each of its four inputs. Model MR168T can do the same for its eight inputs. The transmitter senses the continuity of its input lines. Transmission will occur if a normally open contact is shut or a normally shut contact is opened. With simple modifications of the software, the PIO pins on the DSP board could be used to trigger relays that would be used as the inputs to the transmitter. One input could be used to indicate "I AM OK" while another input would indicate an "ALARM" condition.

This transmitter radiates four watts of power, which is more than the required 1.55 watts determined in Appendix B. It draws less than 20 microamperes in standby and 0.8 amperes during its 4-5 second transmission time. The Linear transmitters can be powered from an external 12-13.5 VDC power supply or nine AAA alkaline batteries. A four-channel

transmitter costs about $240.00 and an eight-channel receiver costs about $300.00.

This family of transmitters and receivers meet most of the design requirements; however, there are some reasons why it is not desired to use these units in the final product. First, the low transmission frequency requires a larger antenna than that needed for high frequency systems (900 MHz). Second, the receivers used with these transmitters can only handle a maximum of eight channels and then only one channel at a time. Finally, signals are not transmitted using spread spectrum methods making it easy to locate the transmitters using Radio Direction Finding (RDF) techniques.

D. POWER SUPPLY

The power supply for this device is a 12 volt, 6.5 Amp-hr, gel-cell, lead-acid battery made by Power Sonic of Redwood City, California. The +12 volt supply is regulated to +5.0 volts for the DSP board. The microphone is supplied by a separate phantom battery source. The transmitter is supplied by either the system 12 volt battery or its own internal AAA batteries. Figure 5 below illustrates the circuit diagram for the +5.0 volt DSP board power supply.

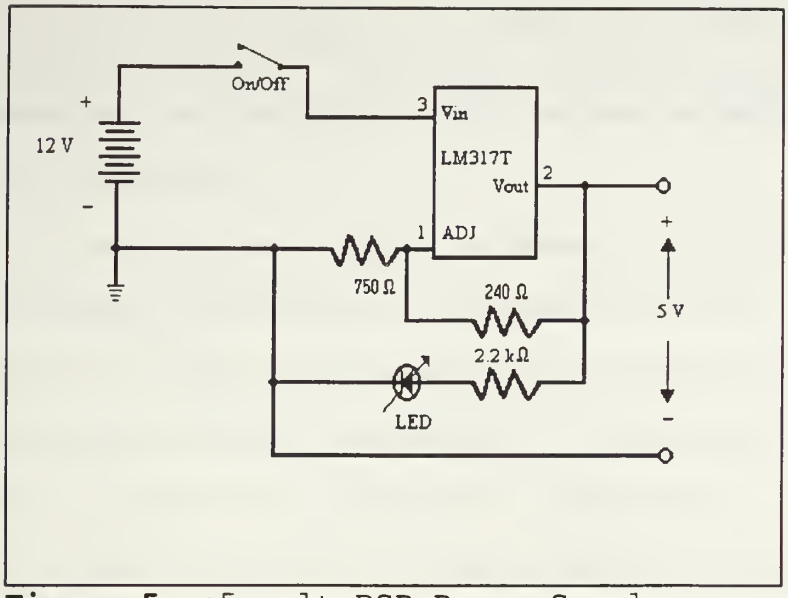


Figure 5 +5 volt DSP Power Supply

E. SUMMARY

After investigating the alternatives for the four hardware categories, the following solutions were chosen for implementation:

- Digital Signal Processing Board: Wavetron Microsystems' EMB-1601A.
- Sensing Devices: Crown PZM-180 microphone, Oyo Geospace geophone model HS-J-K3A.
- Transmitter: Linear's model MR164T four-channel transmitter.
- Power Supply: Alkaline batteries and 12 volt gel-cell lead acid battery.

A system with these components less the transmitter was constructed and tested satisfactorily.

V. SOFTWARE DESIGN CONSIDERATIONS AND IMPLEMENTATION

The software phase of this design was a multi-step process. The first step was to define the problem. Once the problem was defined, a solution algorithm had to be developed. The third stage was to implement the algorithm in code and debug it. Finally, the code was modified to operate independent of the development system and loaded on an EPROM for board testing. This chapter describes the entire design process, and the resulting problem solution.

A. PROBLEM DEFINITION

There are several objectives that must be accomplished in this software design. The problem definition is based on efficiently satisfying all these objectives. Below is a list of the objectives for this design.

1)The overall software structure should be designed so that it meets all of the objectives and is easily expanded by linking in additional C language modules.

2)The code must be loadable onto a boot EPROM for stand-alone operation.

3)The code must cause the DSP board to sample the line or microphone inputs at a user determined frequency.

4)The code must process the sampled data to determine if an "ALARM" condition exists.

5)The code must be able to perform hardware self-checks to determine operability.

6)The code must cause the DSP board to transmit device-unique signals that periodically indicate "I AM OK" or aperiodically indicate that an "ALARM" condition has occurred.

B. SOLUTION ALGORITHM

There are several programming strategies or philosophies that could be used individually or in combination to create a program structure that will meet the above objectives. One such programming strategy is called the Flow Driven method. This method is used when a program needs to step through a sequence of instructions with strict control interrupts and external inputs. This approach is conceptually simple and is good for lockstep type processes. However, it suffers from a lack of flexibility. Consequently, this approach is least suited for real-time applications.

A second approach is called the Clock Driven method. This method is particularly well suited for maintenance type algorithms where a controlling program must call subprograms based on the amount of time expired on some clock. However, the Clock Driven method may be too constraining if the subroutine length is subject to variability or if the number of subroutines is small.

A third approach to program structuring is called the Interrupt Driven method. The Interrupt Driven method uses real-time occurrences to cause the microprocessor to transfer program control to an interrupt handling routine. This method is excellent for real-time applications where immediate action

is required for a given event. Additionally, these interrupts can be nested such that if the processor is servicing one interrupt and a higher priority event comes along, the highest priority event is always serviced immediately at the expense of the lower priority event. When the higher priority event servicing is complete, the processor resumes servicing the lower priority event. One drawback to using this method is that the microprocessor design governs the number and type of interrupts available. This will limit the program's flexibility.

Figure 6 shows a structure chart for the programming algorithm used in this software design. The flow diagrams for the individual blocks of this figure can be found in Appendix B and the associated source code in Appendix C. As can be seen, all three program structuring methods were used.

The Interrupt Driven method was used to handle transferring information to and from the enCOder DECoder (CODEC) peripheral. The Transmit Interrupt (SPORT1 TX) was used during the hardware initialization phase. When the DSP board is RESET, an initialization process is begun. This process sets up the pertinent registers and parameters necessary for the proper operation of the DSP board. At one point in the initialization phase, the CODEC is placed in Control Mode. Four control words are then sent to the CODEC by the microprocessor. These control words specify various operating parameters such as sampling rate, data format, and

stereo/mono mode. After each control word is sent, a SPORT1 TX interrupt is generated. The SPORT1 TX interrupt handler carries out the handshaking routines to ensure the CODEC has been initialized. The interrupt handler then places the CODEC in Data Mode and disables the SPORT1 TX interrupt

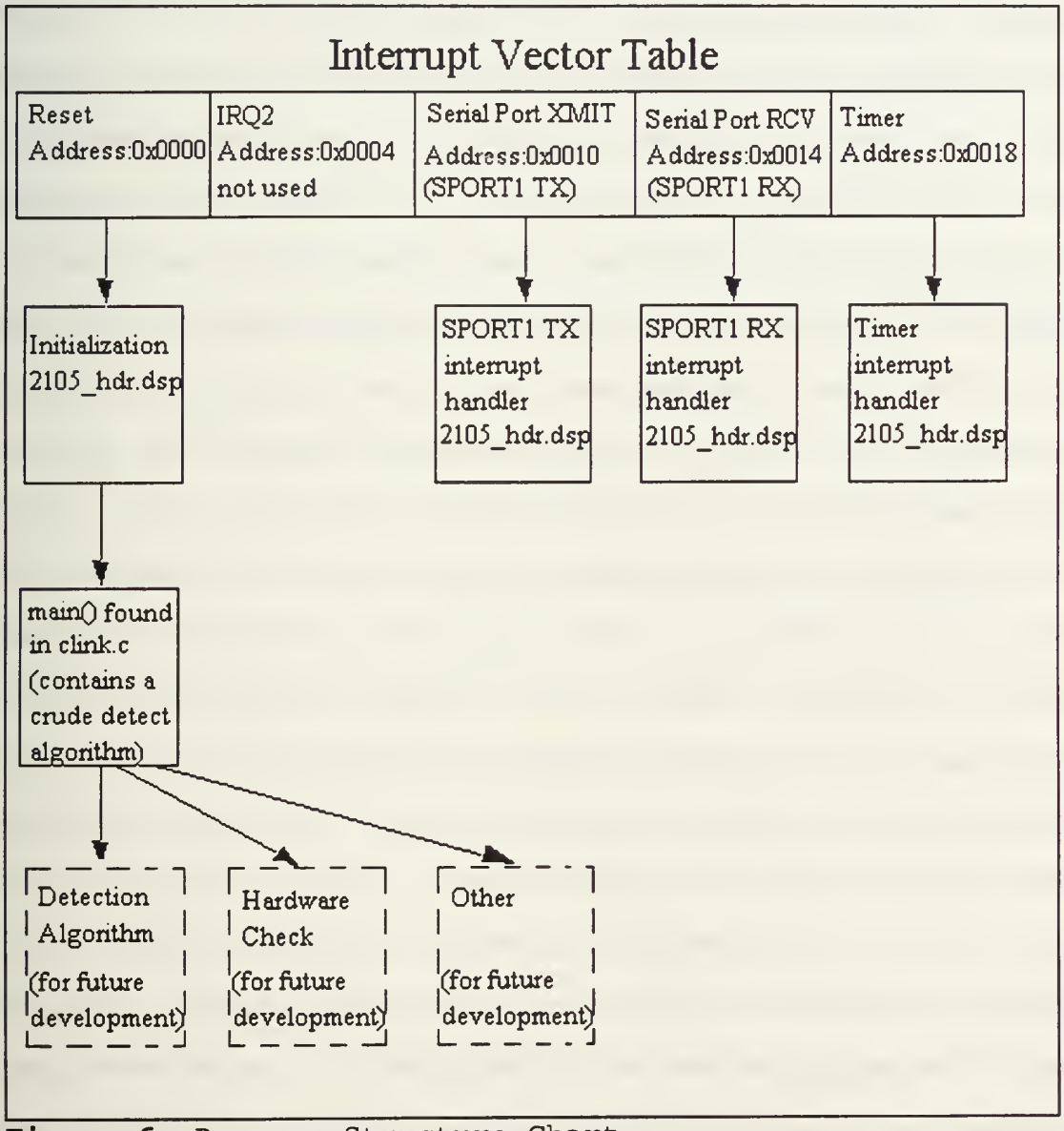


Figure 6 Program Structure Chart

after the CODEC initialization is complete. See Appendix B for a flow diagram describing the SPORT1 TX routine.

Once the CODEC has been placed in Data Mode, the Receive Interrupt (SPORT1 RX) is used to receive and store the sampled data. This interrupt is also used to send data back to the CODEC. The CODEC sends a four-word packet of data to the DSP microprocessor at the sampling rate set while in Control Mode. When the four-word packet is sent to the microprocessor, a SPORT1 RX interrupt is generated by the CODEC. The interrupt handler will do two things. The first thing the SPORT1 RX interrupt handler does is, establish conditions for external signaling. The microprocessor can generate external signals via the CODEC. This is done by properly setting the value of the fourth word in a four word packet (cntrl_1) that is sent to the CODEC in the last part of this interrupt handler. The value of the cntrl_1 comes from one of the three message arrays ("NORMAL", "I AM OK", and "ALARM"). The message array chosen as the source for cntrl_1 is based on the value of the global variable SIGNAL_FLAG. The variable cntrl_1 is assigned the value of a message array element. After a certain number (txn_divisor) of SPORT1 RX interrupts has occurred, cntrl_1 will be assigned the next value in message array. When the end of the array is reached SIGNAL_FLAG will be checked again to determine the appropriate message array. Each entry in a message array determines the voltage level of the two parallel input/output pins (PIO1, PIO0) on the DSP board. A more

detailed explanation of this process can be found in the next section.

The second thing the receive interrupt handler does is read the data from the CODEC inputs. The same data is then immediately written back to the CODEC for output with one exception. It is at this point that a possibly modified cntrl_1 is sent to the CODEC to set the value of the PIO pins. See Appendix B for a flow diagram describing the SPORT1 RX interrupt handler.

The Clock Driven method has been used to periodically set the SIGNAL_FLAG variable to a value indicating an "I AM OK" condition. A timer onboard the microprocessor can be setup to generate an interrupt with a period of up to 1.67 seconds. However, the frequency of this interrupt must be reduced if hardware self-checks are to be conducted *hourly*. This was done by introducing a scaling constant called one_hour. When the one_hour variable is set to 2160, significant action will only be taken on every 2160^{th} Timer interrupt. This means the interrupt handler will take meaningful action about once an hour. The meaningful action for this interrupt is to set the SIGNAL_FLAG variable to a value indicating an "I AM OK" condition. SIGNAL_FLAG will only be set to "I AM OK" if there is no preexisting "ALARM" condition. A flow diagram for the Timer interrupt handler can be found in Appendix B. Future versions of this program could call a hardware checking subroutine from this interrupt and use the results of the

check to set the SIGNAL_FLAG variable to the appropriate value.

The Flow Driven method has been used to initialize the DSP board hardware, initialize the program variables, and call the Main.c program. The initialization is accomplished every time the EMB-1601A is powered up or reset. The Flow Driven method was chosen for the initialization process because the steps of this process must be completed sequentially with strict control of all interrupts. Once all of the initialization has been completed, the Main.c program can be called. Main.c and its subroutines are also flow driven, but they must be interruptable to allow the sampling and signaling interrupt routines to operate. A flow diagram for the initialization process can be found in Appendix B.

C. ALGORITHM ENCODING AND DEBUGGING

The development tools provided by Analog Devices include:

- C Compiler and C Preprocessor.
- ADSP-2100 Family C Runtime Library (floating-point math functions, Digital Signal Processing functions, standard C operations).
- C Source Level Debugging Utility.
- ADSP-2100 Family simulator.
- PROM Splitter

These software tools include a system builder. The System Builder creates an architecture file (*.ach) from a user

written system file (*.sys) that describes the target system being programmed. This architecture file is used by the compiler to help determine where variables and code can be placed. The system file for the EMB-1601A is called 2105.sys and can be found in Appendix C. In addition to the system builder, the software tools use an assembler, linker and a PROM splitter. The PROM splitter converts the executable code into a format that can be loaded onto a PROM.

The Wavetron Microsystems' development package includes:

- A boot PROM with the UART driver and user-code download routine on it.
- Object code and source listing for a Fast Fourier Transform routine.
- C++ functions and a library for initializing and controlling the EMB-1601A from a C++ program.
- An Assembler, Linker, and System builder.

The Wavetron development tools were used exclusively in the early phases of the programming process. The first step in developing the software portion of this system was to create a system file. Once this was done, the next step was to assemble, link, and download an example program provided by Wavetron. The example program consisted of a batch program (*.BAT), several Analog Devices' assembly language modules (*.DSP), and a C++ program (*.cpp). The batch file assembles, links, and converts the object code to a format that can be downloaded from the PC to the DSP board. The C++ program is

used to initialize the DSP board, establish a graphical interface, and download the user-code onto the DSP board. Once the C++ program downloads the DSP code onto the DSP board, it receives data from the DSP board which is used to display the frequency spectrum of the input signals on the PC. The assembly language modules setup the interrupt tables, perform an FFT on two incoming line signals and make the data available to the PC for graphing on the PC monitor. Structure charts for these programs are shown Figure 7. The ability to graphically display the spectrum of sampled data made these routines ideal for monitoring the response of the seismic and acoustic sensors. All of the source files for this example program can be found on the development software disk provided by Wavetron Microsystems.

After the example program was running successfully, the next step was to modify the program so that the voltage levels at the PIO pins could be changed. As explained in the previous section, this is accomplished by modifying the value of the data words that are being sent to the CODEC. By changing the value of the PIO bits in cntrl_1, the level of the PIO pins can be changed. Figure 8 shows the format of the four-word, 48-bit data stream that is sent to and received from the CODEC. It should be noted that cntrl_1 and cntrl_2 are subsets of the data stream and are treated as variables in the programs. The variable cntrl_1 was modified in the receive interrupt (SPORT1 RX). This modification was scaled

to occur periodically after a set number of interrupts had occurred. The scaling was done by setting the variable txn_divisor to some desired value. A value of 5000 for txn_divisor causes a new value to be sent to the PIO pins at a rate of once every two seconds. This slowed rate permits visual monitoring of the changing PIO pin values by means of an LED display.

Once the mechanism for changing the PIO pin values was proven to work, the next step was to change the PIO levels using the Timer interrupt. The timer's initial count, scaling value, and reload values are set by writing the appropriate values to the memory mapped Timer Registers. By writing to these registers, the timer can be setup to generate a periodic interrupt as described above.

The program was then modified to transmit two different signals to the PIO pins. The first signal indicates an "I AM OK" condition as generated by the Timer interrupt routine. The second signal indicates an "ALARM" condition. The alarm is periodically generated by setting SIGNAL_FLAG to the "ALARM" value at the end of a large infinite loop in the main body of the program. The Receive interrupt (SPORT1 RX) handler periodically checks the SIGNAL_FLAG variable to determine the message to be sent. Then the interrupt handler sets the cntrl_1 variable equal to the first value of the appropriate message array and resets the SIGNAL_FLAG variable

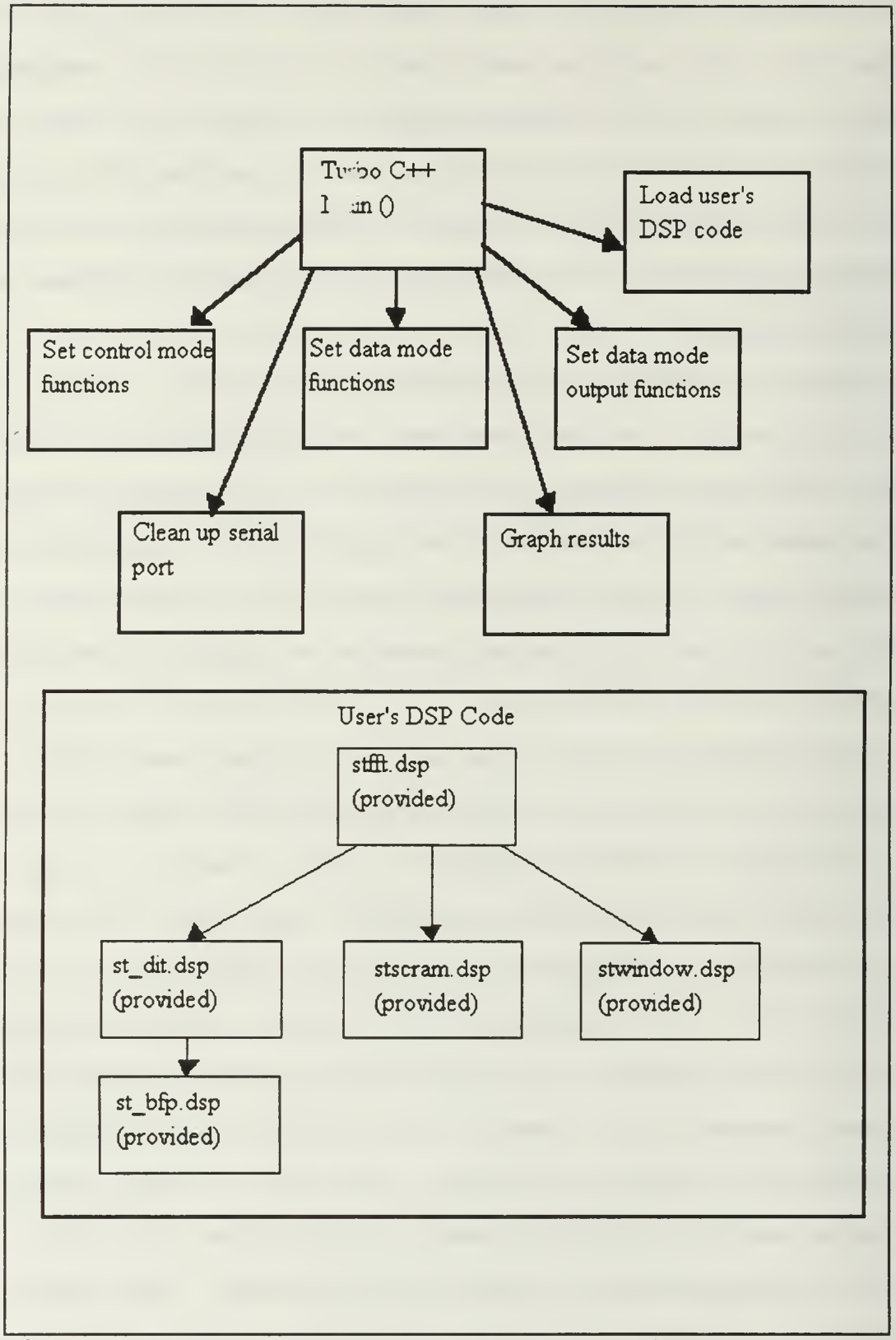


Figure 7 Structure Charts for the Example Program

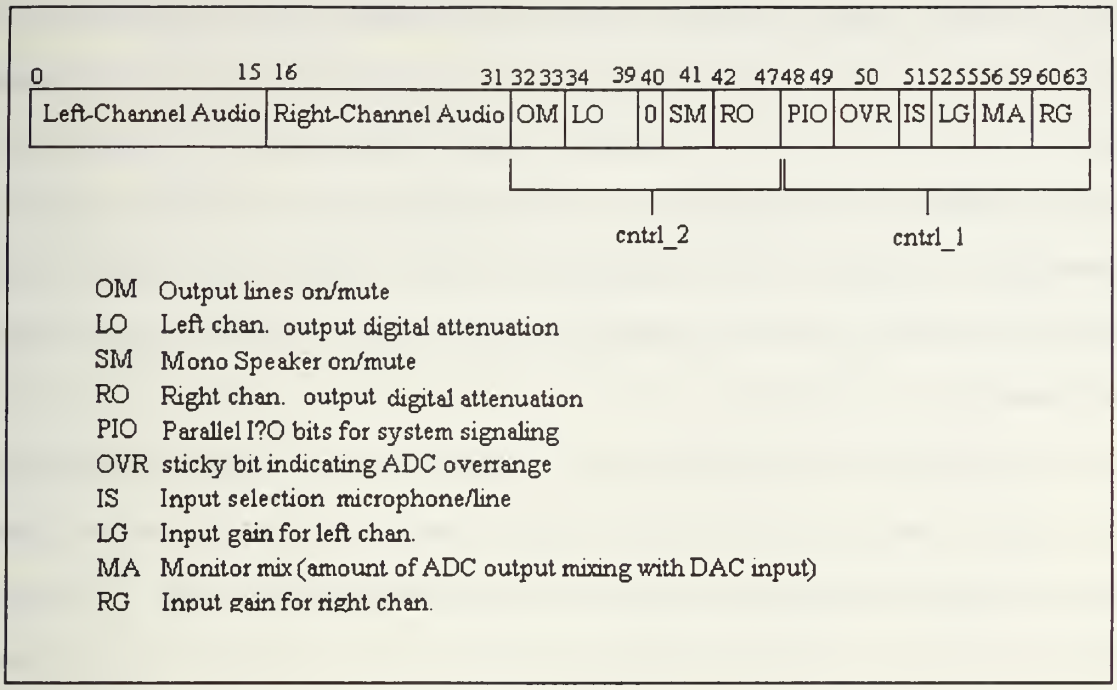


Figure 8 16-Bit Stereo Data Word

to indicate a NORMAL condition. After txn_divisor interrupts occurred, the cntrl_1 variable is set to the next value in the message array. When the end of the array has been reached, the SPORT1 RX handler reexamines the value of the SIGNAL_FLAG variable and the process begins again. The bit pattern of the messages can be easily modified to indicate the identity of different UGSs by changing the message array values in the source code. The code can also be easily modified for operation with the Linear transmitters described in Chapter IV. The source code for this step can be found in file TRANSMIT.DSP listing in Appendix C.

After this algorithm was successfully encoded in Analog Devices' Assembly language and debugged, the next step was to link a C-language program with it. The C-program was kept

simple and was used to set the SIGNAL_FLAG variable to indicate an alarm condition when the geophone input exceeded a set threshold. This C-code can be found in the listing for file CLINK.C in Appendix C. Linking the C-code with the assembly language code proved to be quite difficult and required considerable technical help from an Analog Devices' programming consultant.

One of the difficulties in linking the programs stemmed from the fact that the ADSP-2100 Family of processors have five different memory spaces. Boot memory is one of these memory spaces. The Boot Memory (BM) is split into eight pages and each page is one-thousand words long for an ADSP-2105 processor. In addition to BM, there are internal (on chip) and external Data Memory (DM) spaces. Program Memory (PM) is also split into internal and external sections. The size and location of these memory spaces must be specified in the system file as mentioned earlier. One of the problems was due to the fact that there was no way to dictate where a compiled C-program was to be loaded. The G21 compiler provided by Analog Devices automatically placed the C-code at address 0x0000 in internal PM. Since the code on the Wavetron Microsystems' PROM also specified that the download routines must be placed at address 0x0000 of internal PM, there was a conflict in the memory spaces. This conflict prevented loading C-linked code onto the DSP board from the PC. It was

determined that there was no solution to get around this problem.

However, there were still two ways to test the code. The first method was to run the code on the Analog Devices' system simulator. The code tested by this method ran as expected. Nevertheless, the only reliable way to test the code with C modules linked in is to write the code for stand-alone processor operation and load the code onto an EPROM. The EPROM can then be installed on the DSP board and the code tested. The process for doing this is found in the next section.

D. MODIFICATIONS FOR STAND-ALONE OPERATION

This section describes the next step in the software development. After the code was tested either by simulation or downloading it onto the DSP board by means of the Wavetron Microsystems' utility programs, it was loaded onto an EPROM and tested on the DSP board in a stand-alone mode.

This process required modifications to the assembly language code. The EPROM would replace the Wavetron Microsystems' PROM which had setup the interrupt table vectors. This meant the stand-alone code must supply its own interrupt table settings. It also meant that the modified code had to be placed at address 0x0000 in PM. This would ensure the interrupt table was in the proper location. With these modifications made to the previously developed code, the

new code was then compiled and the executable (*.exe) code obtained. The PROM Splitter was then invoked to convert the executable code into a format suitable for booting. The specific format of this bootable version is selectable by the proper use of switches in the PROM Splitter (SPL21) command line. The bootable code was then downloaded onto a 27C512 EPROM for DSP board testing. Appendix D contains procedures that outline all of the steps that must followed in converting a user's C-code to a programmed EPROM.

An EPROM was programmed at the end of each major stage of the software design process to test the code on the DSP board. The C-linked program was also loaded onto an EPROM and tested satisfactorily. This successful test proved the ability to link the user's C-code with the interrupt handlers.

E. SUMMARY

The process to develop the code for this project involved several steps. The first step was to successfully load the example code provided by Wavetron Microsystems onto the DSP board. The next major step was to modify this code to prove that the PIO pins could be manipulated for transmission. The third step used the Timer interrupt and a large infinite loop to select the type of signal to be transmitted. The final step was to develop C-code that would examine the input data, check for an alarm condition, and set the SIGNAL_FLAG accordingly. Each of these steps were tested using a software

simulator. The code was then verified on the DSP board by downloading it to the board via the PC or programming a boot EPROM with the code.

There are only two ways to test code written in the C programming language. The first method is to use the software simulator provided by Analog Devices. The second method is to link the C code with its assembly language header file and load this file onto a boot EPROM. Once the boot EPROM is installed on the DSP board, the code begins execution at power up or when the reset switch is depressed.

VI. CONCLUSIONS AND RECOMMENDATIONS

The Unattended Ground Sensor described in this thesis was the best solution that could be provided with the resources that were available at the time. The UGS in this design uses a digital signal processing board manufactured by Wavetron Microsystems to sample acoustic and seismic sensors. The acoustic sensor is a microphone made by Crown and has a hemispherical beam pattern. The seismic sensor is a small, lightweight geophone made by Oyo Geospace.

The data from these sensors will be processed using DSP algorithms that are to be developed. The DSP code developers will be able to initiate an ALARM transmission simply by setting the global variable SIGNAL_FLAG to a number greater than zero. The software written for this project will carry out the details of transmitting the ALARM signal.

A lack of funding prevented obtaining a transmitter for incorporation in this design. However, a transmitter was identified for temporary use. The transmitter chosen is made by Linear and is typically used in burglar alarm systems. The drawbacks of this transmitter are that it does not transmit in spread spectrum mode and it transmits at too low of a frequency. It is recommended that a small, high-frequency,

spread spectrum transmitter be developed specifically for use in this project.

The major problem with the UGS designed for this project is its high power consumption. The bulk of the power is drawn by the DSP board itself. This problem was unavoidable in this design due to the requirement to use commercially available of-the-shelf technology. There are some DSP systems currently being developed that use very low power. One of these systems is the Underwater Digital Signal Processor (UDSP) made by Mikros Systems Corporation. The UDSP is based on the Allied Signals 1750-A microprocessor. It can compute at up to four million instructions per second (MIPS) while consuming only one watt of power. At five milliwatts it can carry out computations at 10,000 operations per second. It is recommended that low power devices like this one be investigated further for use in future UGS designs.

This UGS, as designed, is a good platform for developing and testing digital signal processing algorithms. Once these algorithms have been shown to be effective, the UGS portion of the UAV/UGS combination concept will have been proven.

APPENDIX A: DSP BOARD SUMMARY

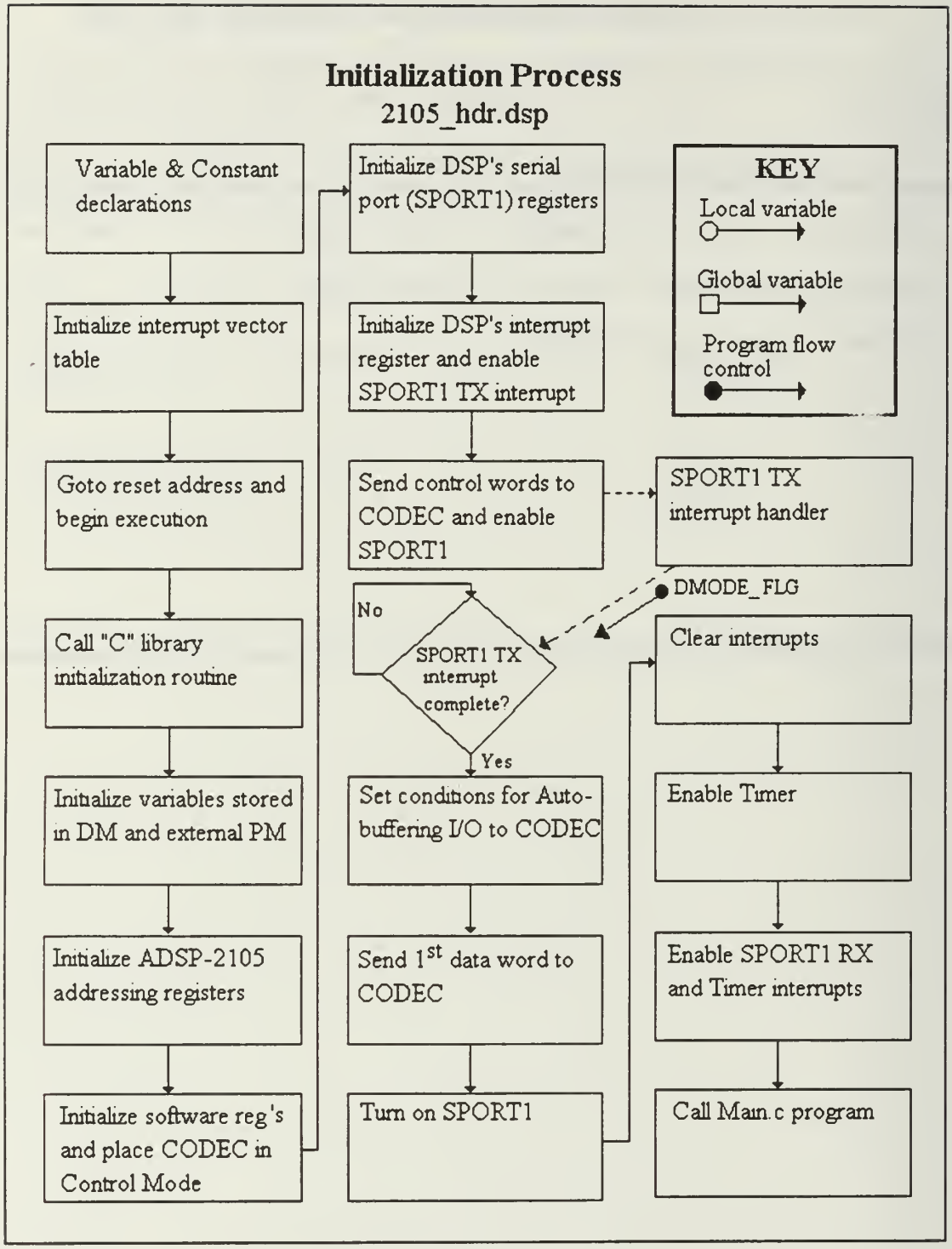
Table II DIGITAL SIGNAL PROCESSOR COMPARISON CHART

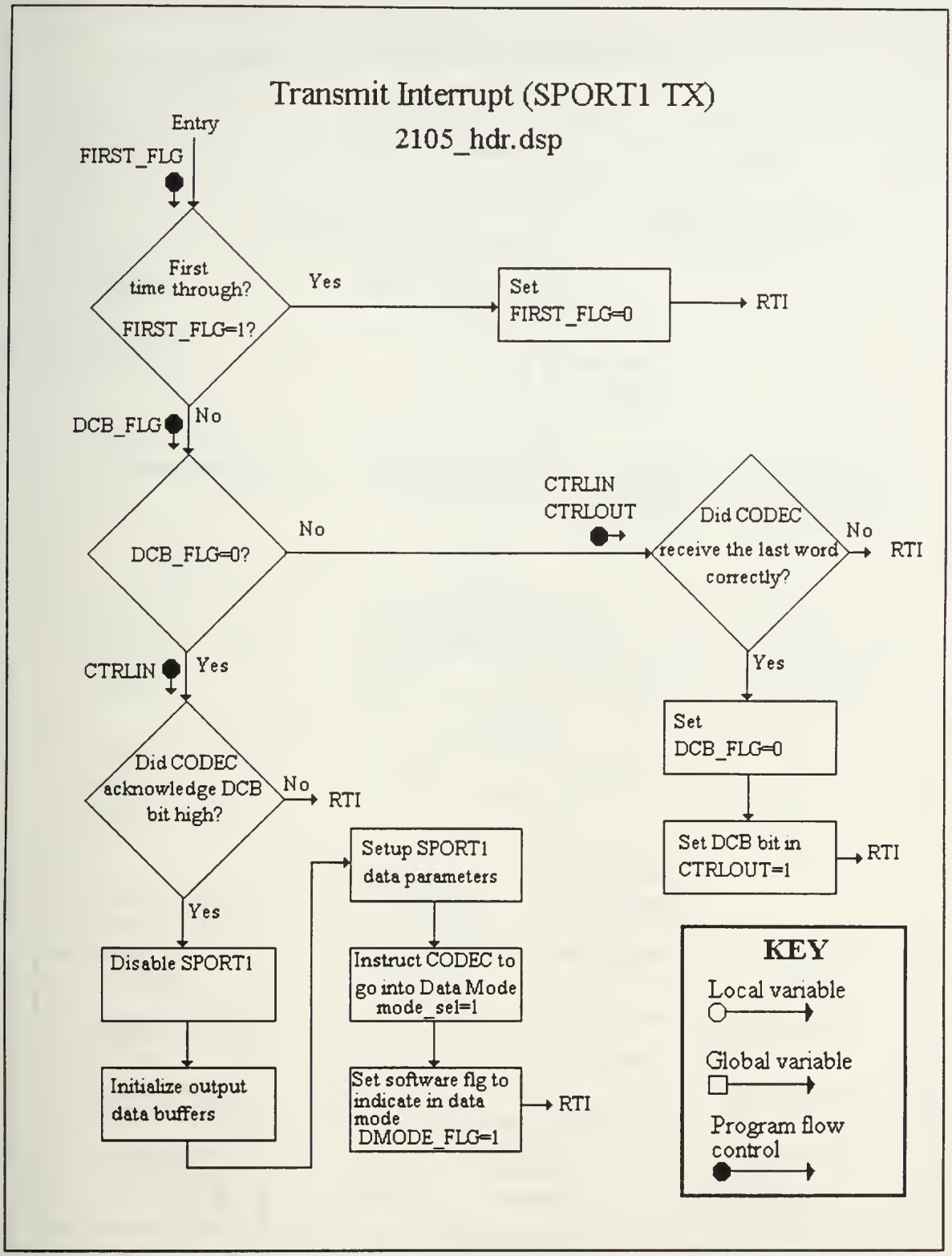
MFR/MODEL	CHIP	COST	POWER	I/O
DSP Research/Piranha 3111	TMS320C31	1500.00	2 W	1-analog I/O
Innovative Integration/SBC-31	TMS320C31	1150.00	3 W	2-RS-232 chan 16-analog in 4-analog out 48-bit digital I/O
Wavetron Microsystems/EMB-1601 (with optional memory)	ADSP-2105	590.00	2 W	2-line in 2-line out 1-mic in 2-PIO pins

APPENDIX B: FLOW DIAGRAMS FOR FILE 2105_HDR.DSP

This Appendix contains the flow diagrams for the file 2105_HDR.DSP. This file is broken down into the following sections:

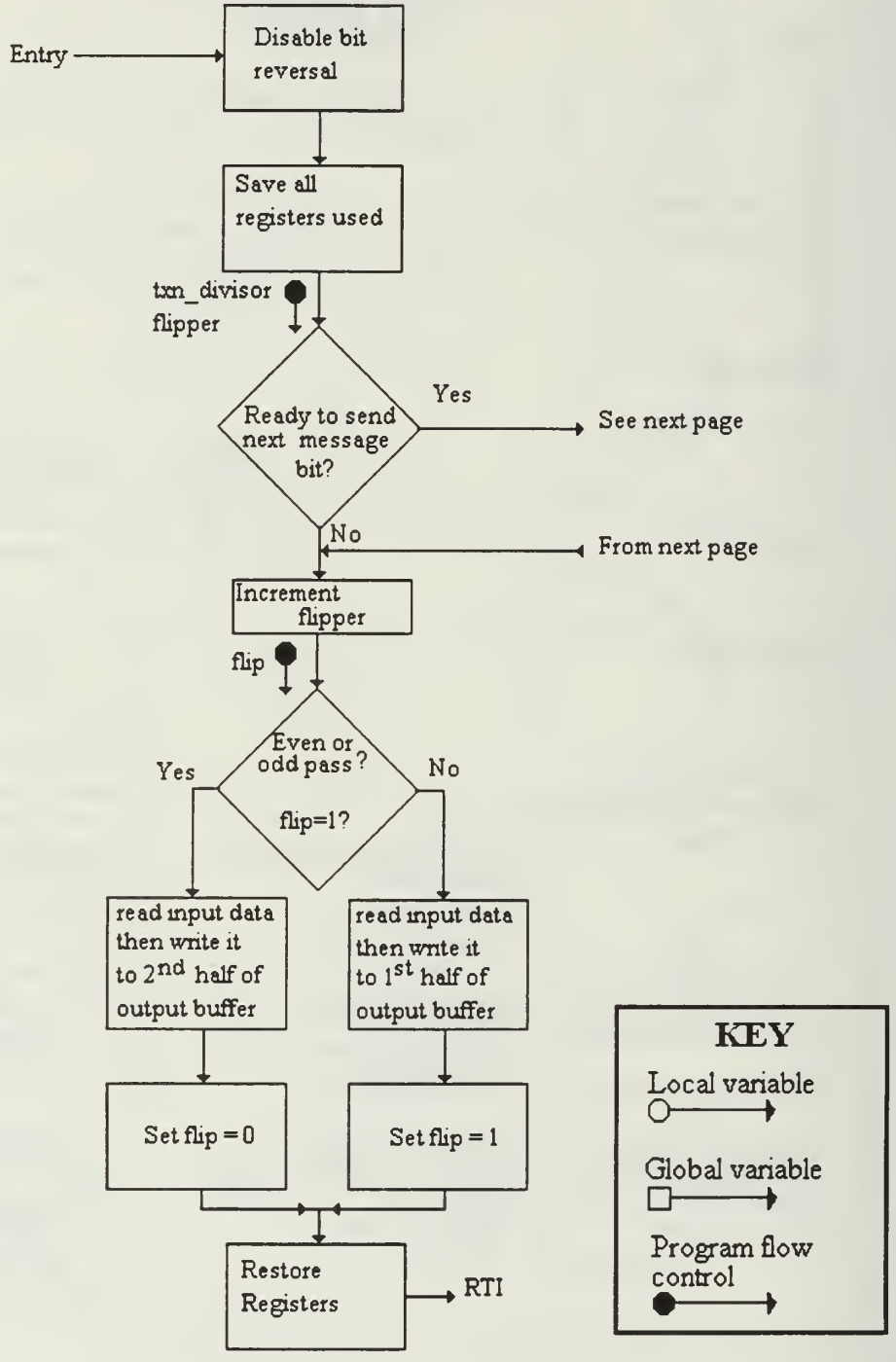
- Initialization Process
- Transmit Interrupt Handler (SPORT1 TX)
- Receive Interrupt Handler (SPORT1 RX)
- Timer Interrupt Handler

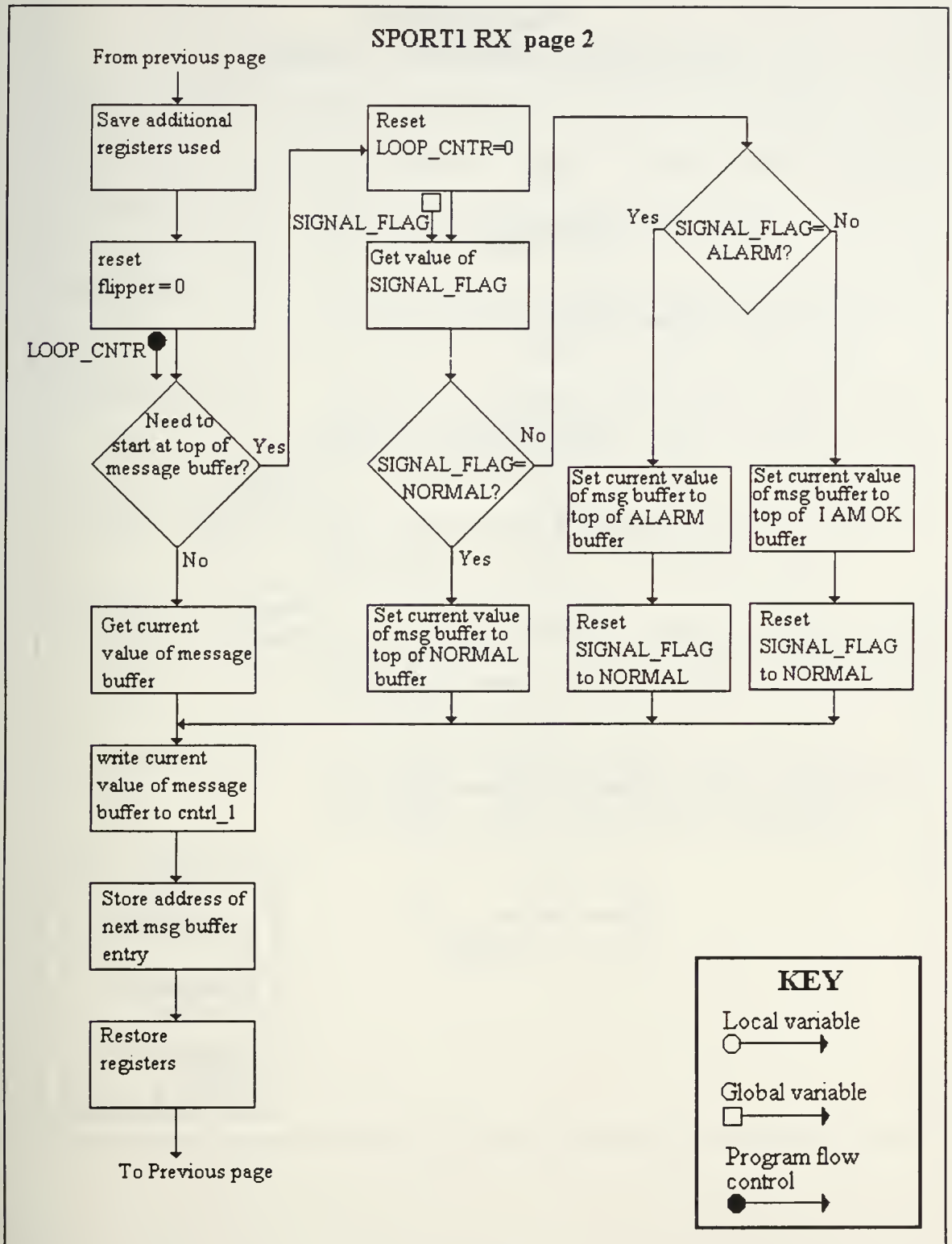

Initialization Process
2105_hdr.dsp
Variable & Constant declarations
Initialize interrupt vector table
Goto reset address and begin execution
Call "C" library initialization routine
Initialize variables stored in DM and external PM
Initialize ADSP-2105 addressing registers
Initialize software reg's and place CODEC in Control Mode
Initialize DSP's serial port (SPORT1) registers
Initialize DSP's interrupt register and enable SPORT1 TX interrupt
Send control words to CODEC and enable SPORT1
No
SPORT1 TX interrupt complete?
Yes
Set conditions for Auto-buffering I/O to CODEC
Send 1st data word to CODEC
Turn on SPORT1
KEY
Local variable
Global variable
Program flow control
SPORT1 TX interrupt handler
DMODE_FLG
Clear interrupts
Enable Timer
Enable SPORT1 RX and Timer interrupts
Call Main.c program

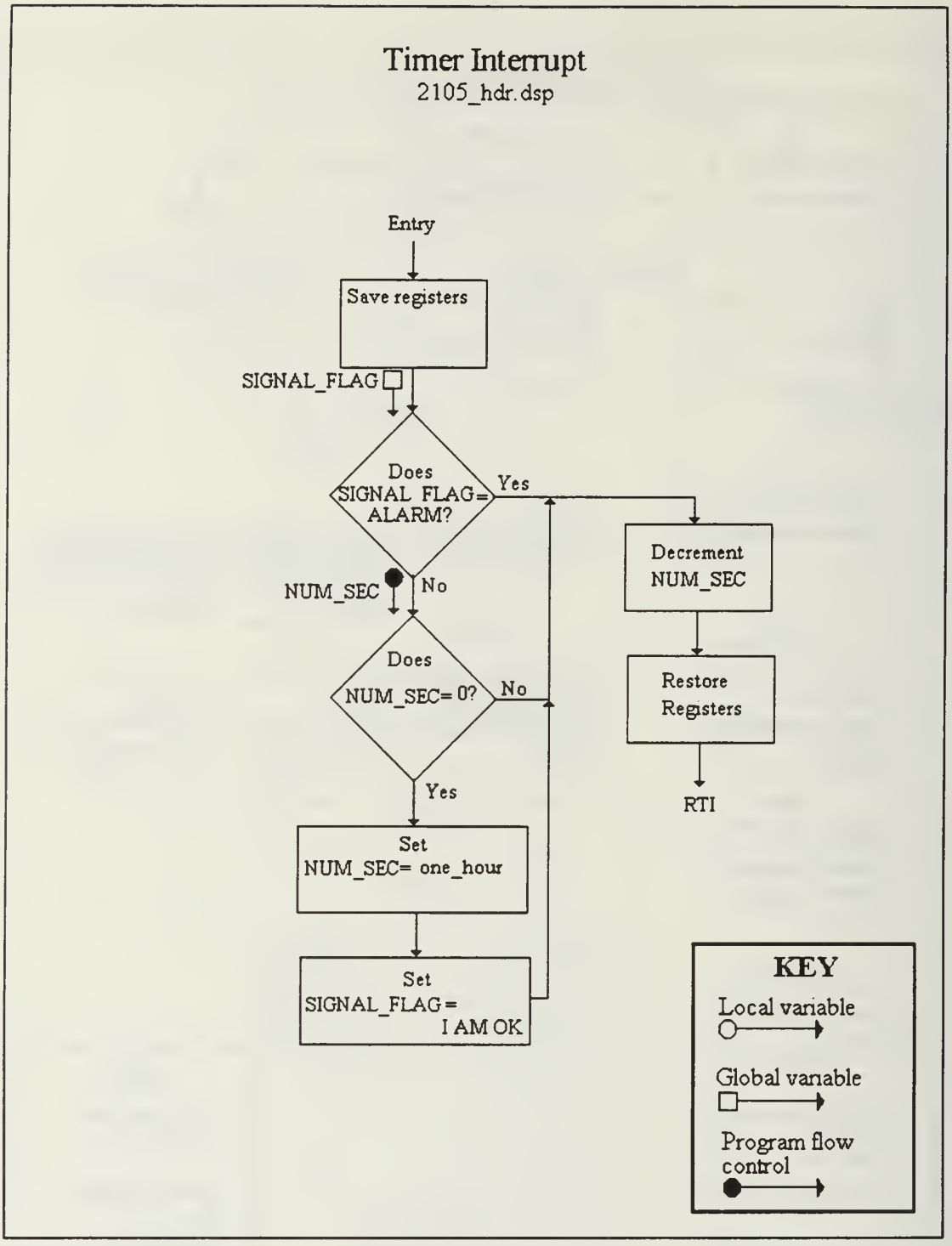

Transmit Interrupt (SPORT1 TX)
2105_hdr.dsp
Entry
FIRST_FLG
First time through? FIRST_FLG=1?
Yes
Set FIRST_FLG=0
RTI
DCB_FLG
No
DCB_FLG=0?
No
CTRLIN
CTRLOUT
Did CODEC receive the last word correctly?
No
RTI
Yes
Set DCB_FLG=0
Set DCB bit in CTRLOUT=1
RTI
CTRLIN
Yes
Did CODEC acknowledge DCB bit high?
No
RTI
Yes
Setup SPORT1 data parameters
Disable SPORT1
Instruct CODEC to go into Data Mode mode_sel=1
Initialize output data buffers
Set software flg to indicate in data mode DMODE_FLG=1
RTI
KEY
Local variable
Global variable
Program flow control

Receive Interrupt (SPORT1 RX)

2105_hdr.dsp




SPORT1 RX page 2
From previous page
Save additional registers used
reset flipper = 0
LOOP_CNTR
Need to start at top of message buffer?
Yes
No
Get current value of message buffer
write current value of message buffer to cntrl_1
Store address of next msg buffer entry
Restore registers
To Previous page
Reset LOOP_CNTR=0
SIGNAL_FLAG
Get value of SIGNAL_FLAG
SIGNAL_FLAG= NORMAL?
No
Yes
Set current value of msg buffer to top of NORMAL buffer
Yes
SIGNAL_FLAG= ALARM?
No
Set current value of msg buffer to top of ALARM buffer
Reset SIGNAL_FLAG to NORMAL
Set current value of msg buffer to top of I AM OK buffer
Reset SIGNAL_FLAG to NORMAL
KEY
Local variable
Global variable
Program flow control


Timer Interrupt
2105_hdr.dsp
Entry
Save registers
SIGNAL_FLAG
Does SIGNAL_FLAG= ALARM?
Yes
Decrement NUM_SEC
NUM_SEC
No
Does NUM_SEC= 0?
No
Restore Registers
RTI
Yes
Set NUM_SEC= one_hour
Set SIGNAL_FLAG= I AM OK
KEY
Local variable
Global variable
Program flow control

APPENDIX C: PROGRAM SOURCE CODE

This appendix contains listings of the following files:

- 2105.SYS — Description file for the system builder.
- FREETEST.BAT — DOS batch file that assembles, links, and PROM splits TRANSMIT.DSP.
- PIO_TEST.CPP — C++ file used to initialize the PC for communication with the DSP board and send the user code to the DSP board.
- TRANSMIT.DSP — Assembly language code that tests the use of the Timer interrupt to send an "I AM OK" signal and a large infinite loop to send an "ALARM" signal.
- C-LINK.BAT — DOS batch file that assembles, links, and PROM splits 2015_HDR.DSP and CLINK.C.
- CLINK.C — C file that compares sampled data to a threshold value and sets the SIGNAL_FLAG to the "ALARM" value if the threshold is exceeded.
- 2105_HDR.DSP — Assembly language file that initializes the DSP board, contains the interrupt handlers, and calls the C program.

2105.SYS

```
{Description file for the System Builder specifies the
amount of data and program memory in the system.  The file
also declares each page of boot memory which will be used.}

.SYSTEM dsps;                          {system name}
.ADSP2105;                             {specifies processor}
.MMAP0;                                {boot loading enable}
.SEG/ROM/BOOT=0 BOOT_0[1024];          {boot page zero}
.SEG/ROM/BOOT=1 BOOT_1[1024];          {boot page one}
.SEG/ROM/BOOT=2 BOOT_2[1024];
.SEG/ROM/BOOT=3 BOOT_3[1024];
.SEG/ROM/BOOT=4 BOOT_4[1024];
.SEG/ROM/BOOT=5 BOOT_5[1024];
.SEG/ROM/BOOT=6 BOOT_6[1024];
.SEG/ROM/BOOT=7 BOOT_7[1024];          {boot page seven}

{internal program memory at absolute address 0x0000, 1024
words long}
.SEG/PM/ram/abs=0/code/data int_pm[1024];

{external program memory 4096 words long.  External program
memory and external data memory together cannot exceed 8192
words}
.seg/pm/ram/abs=12288/code/data ext_pm[4096];
.seg/dm/ram/abs=0x1000/data mode_select;

{external data memory 4096 words long.}
.seg/dm/ram/abs=8192/data ext_dm[4096];

{internal data memory 512 words long.}
.seg/dm/ram/abs=14336/data int_dm[512];
.endsys;
```

FREETEST.BAT

DOS batch file that assembles, links, and PROM splits transmit.dsp. Transmit.dsp tests the software's ability to choose and send either an "ALARM" or "I AM OK" signal. EMB_BOOT.DSP contains the interrupt table setup and routines for downloading code from the PC. See Reference 10 for a listing of EMB_BOOT.DSP.

```
erase emb_send.dat
asm21 transmit.dsp -l
asm21 emb_boot.dsp -l
ld21 emb_boot transmit -a 2105 -lib -x -g -e 210x
spl21 210x emb_boot -i -bm
emb_load emb_send.dat
```

PIO_TEST.CPP

The following file is a C++ file used to initialize the PC for communication with the DSP board and send the user code to the DSP board.

```
/* Embedded board test */
#include "emb_head.hpp"
#include <stdio.h>
#include <conio.h>
#include <dos.h>
#include <math.h>
#include <iostream.h>

void main (void)
{
    clrscr();
    init_eb1601(1);         /* Initialize EMB1601 on COM2 */

    load_code("emb_send.dat");    /* Load user's DSP code */
    wait_1(5000);              /* Wait 'til code is running */

    cleanup_com();             /* Reset serial port */
}
```

TRANSMIT.DSP

The following file contains code that uses the Timer interrupt to periodically set conditions for sending an "I AM OK" signal. It also utilizes an infinite loop in the main body to establish conditions for sending an "ALARM" signal.

```
.MODULE/RAM/ABS=0xBE/BOOT=0 code_dsp; {module name and load
                                        location}
.CONST SYS_CTL_REG=0x3FFF; (system control register}
.CONST mode_sel=0x1000; {latched control for Control / Data
                         line }
{*********************************************************}
{*********   Variable Declaration    **********************}
.VAR/RAM/CIRC  CTRLIN[4];{circular buffers for data input
                          and}
.VAR/RAM/CIRC  CTRLOUT[4];    {output for data mode and
                               control mode}
.VAR/RAM/CIRC  DATAIN[4];
.VAR/RAM/CIRC  DATAOUT_[8];
.var/ram/circ  ALARM[16];  {signal buffers should all be the
                            same length}
.var/ram/circ  IAMOK[16];
.var/ram/circ  NORMAL[16];
.var/ram       signal_flag, LOOP_CNTR;
.VAR/RAM       FIRST_FLG; {First time thru flag}
.VAR/RAM       DCB_FLG; {DCB software handshaking flag
                         between 1849 & 2105}
.VAR/RAM       DMODE_FLG; {1849 mode flag i.e. CONTROL or
                           DATA modes}
.VAR/RAM       cntrl_1,cntrl_2, AR_save, AX0_save, flip,
                flipper, AY1_save;
.VAR/RAM       AX0H_save, AY1H_save, NUM_SEC, num_samples,
                data_val_;
.CONST one_hour=17;      {1:1.67 sec ratio @ 5 kHz sampling,
                         2160=1 hour}
.CONST txn_divisor=2000;  {used to slow rate at which bits
                           are transmitted, 2000 = 1bit
                           every 0.4 sec when sampling at
                           5 Khz}
.CONST threshold=0X00FF;   {threshold for incoming signal
                           magnitude}
.CONST min_samples=300;   {this is the min number of samples
                           to ensure}
                           {valid data is available.}

{ The following statements make these labels visible to
EMB_BOOT.DSP}
.ENTRY code_start;
.ENTRY irq2_intr;
```

```
.ENTRY irq1_intr;
.ENTRY irq0_intr;
.ENTRY timer_intr;
.ENTRY sport_txm_intr;
.ENTRY sport_rec_intr;

{================ INTERRUPT VECTORS ======================}
code_start:  JUMP through;        {goto through: at reset}
irq2_intr: RTI;                   {Sample clock interrupt}
irq1_intr: JUMP SETUPCONTROL;    {transmit interrupt}
irq0_intr: JUMP NEWDATA;          {Receive interrupt}
timer_intr: JUMP HOUR;          {Internal timer interrupt}
sport_txm_intr: RTI;     {Serial port transmit interrupt}
sport_rec_intr: RTI;     {Serial port receive interrupt}


set_control:     {see EMB-1601A users manual for a detailed
                  description of the control words}
    AX1 = 0x212C;    {samp freq=44.1 khz,stereo,pcm16,2104}
    DM(CTRLOUT) = AX1;
    AX1 = 0x2200;     {xtal2,64bits/frame,master,serial txn}
    DM(CTRLOUT+1) = AX1;
    AX1 = 0xC0F0;              {(PIO=11, etc) Data mode
                        parameter defaults}
    DM(cntrl_1) = AX1;
    AX1 = 0xC000;    {OM, etc}
    DM(cntrl_2) = AX1;
    RTS;

reset_flipper:
    AX0 = 0;          {reset flipper and assign or step
                       buffer}
    DM(flipper) = AX0;
    AY1 = DM(LOOP_CNTR);  {LOOP_CNTR initialized at reset}
    AR = AY1 +1;
    DM(LOOP_CNTR) = AR;
    AX0 = AR;
    AY1 = %IAMOK;       {get length of a message buffer.
           Could have used the length of any message buffer}
    AR = AX0 - AY1;
    IF LT JUMP FINISH;
         AX0 = 0;
         DM(LOOP_CNTR) = AX0; {reset LOOP_CNTR}
         AX0=DM(signal_flag); {get value of signal_flag}
         AR=PASS AX0;
         IF EQ JUMP NORMAL_XMIT;  {are conditions normal?}
         IF GT JUMP ALARM_XMIT;  {is there an alarm?}
IAMOK_XMIT:  I5=^IAMOK;  {set up transmitting IAMOK}
         L5=%IAMOK;
         AX0=0;      {return signal_flag to normal after
                      transmit conditions set}
```

```
          DM(signal_flag)=AX0;
          JUMP FINISH;

NORMAL_XMIT:    I5=^NORMAL; {establish conditions for}
          L5=%NORMAL;       {no transmission}
          JUMP FINISH;

ALARM_XMIT:  I5=^ALARM;  {set up for transmitting an}
          L5=%ALARM;      {alarm}
          AX0=0;          {return signal_flag to normal after
                           transmit conditions set}
          DM(signal_flag)=AX0;

FINISH:
     AX0 = DM(I5,M7);
     DM(cntrl_1) = AX0;
     RTS;


fill_mssg_buffer:
     M7=1;           {fill buffers for auto transmit}
{===MSB is the transmit bit (pio1).  MSB-1 is the data bit
(pio0). ========}
     I5=^ALARM;        {id=0x0253, alarm=lsb=1}
     L5=0;
     DM(I5,M7)=0X00F0;        {P0}
     DM(I5,M7)=0XC0F0;        {C0}
     DM(I5,M7)=0X00F0;        {C1}
     DM(I5,M7)=0XC0F0;        {D1}
     DM(I5,M7)=0X00F0;        {C2}
     DM(I5,M7)=0XC0F0;        {D2}
     DM(I5,M7)=0X00F0;        {D3}
     DM(I5,M7)=0XC0F0;        {D4}
     DM(I5,M7)=0X00F0;        {C3}
     DM(I5,M7)=0XC0F0;        {D5}
     DM(I5,M7)=0X00F0;        {D6}
     DM(I5,M7)=0XC0F0;        {D7}
     DM(I5,M7)=0X00F0;        {D8}
     DM(I5,M7)=0XC0F0;        {D9}
     DM(I5,M7)=0X00F0;        {D10}
     DM(I5,M7)=0XC0F0;        {ALARM BIT}

     I5=^IAMOK;
     L5=0;
     DM(I5,M7)=0X40F0;        {P0}
     DM(I5,M7)=0X80F0;        {C0}
     DM(I5,M7)=0X40F0;        {C1}
     DM(I5,M7)=0X80F0;        {D1}
     DM(I5,M7)=0X40F0;        {C2}
     DM(I5,M7)=0X80F0;        {D2}
     DM(I5,M7)=0X40F0;        {D3}
```

```
    DM(I5,M7)=0X80F0;          {D4}
    DM(I5,M7)=0X40F0;          {C3}
    DM(I5,M7)=0X80F0;          {D5}
    DM(I5,M7)=0X40F0;          {D6}
    DM(I5,M7)=0X80F0;          {D7}
    DM(I5,M7)=0X40F0;          {D8}
    DM(I5,M7)=0X80F0;          {D9}
    DM(I5,M7)=0X40F0;          {D10}
    DM(I5,M7)=0X80F0;          {ALARM BIT}

    I5=^NORMAL;
    L5=0;
    DM(I5,M7)=0X40F0;
    DM(I5,M7)=0X40F0;        {ensures transmitter is off}
    DM(I5,M7)=0X40F0;
    DM(I5,M7)=0X40F0;
    DM(I5,M7)=0X40F0;
    DM(I5,M7)=0X40F0;
    DM(I5,M7)=0X40F0;
    DM(I5,M7)=0X40F0;
    DM(I5,M7)=0X40F0;
    DM(I5,M7)=0X40F0;
    DM(I5,M7)=0X40F0;
    DM(I5,M7)=0X40F0;
    DM(I5,M7)=0X40F0;
    DM(I5,M7)=0X40F0;
    DM(I5,M7)=0X40F0;
    DM(I5,M7)=0X40F0;
    RTS;


through:
    IFC=H#003F;{CLEAR ALL PENDING INTERRUPTS}
    IMASK=H#0000;

{**********  initialize variables and registers  **********}

    CALL set_control; {sets values of control words}
    AR = %IAMOK-1;     {initialize LOOP_CNTR to the length
                        of a message array - 1}
    DM(LOOP_CNTR) = AR;
    AX0 = txn_divisor;
    DM(flipper) = AX0;
    AX0 = 1;
    DM(flip) = AX0;
    AX0=0;
    DM(num_samples) = AX0;
    DM(data_val_) = AX0;
    DM(mode_sel) = AX0;
    DM(signal_flag) = AX0;
    AX0 = 0xFFFF;
```

```
    DM(0x3FFD) = AX0;        {load TPERIOD}
    DM(0x3FFC) = AX0;       {load TCOUNT}
    AX0 = 0xFF;
    DM(0x3FFB) = AX0;        {load TSCALE}
    AX0 = one_hour;
    DM(NUM_SEC) = AX0;
{====Initialize the transmit buffers=======================}
    CALL fill_mssg_buffer;
{====Initilaize the addressing registers of 2105===========}
    L7=%CTRLIN;
    I7=^CTRLIN;
    L6=%CTRLOUT;
    M1=1;
    M7=1;
    I6=^CTRLOUT;
{====Initialize software flags=============================}
    AX0=1;
    DM(FIRST_FLG)=AX0;
    DM(DCB_FLG)=AX0;
    AX0=0;
    DM(DMODE_FLG)=AX0;       { in control mode }
    AY0=DM(CTRLOUT);

{====Initialize the DSP's SPORT1 Serial port registers=====}

     L2 = 0;              { linear addressing for register }
     I2 = 0x3fef;           { point to last DM cntrl reg }
     DM(I2,M1) = 0x0DFF;    { I6,M7,I7,M7 sport1 autobuffer
register }
     DM(I2,M1) = 383;         { rfsdiv1   }
     DM(I2,M1) = 849;         { sclkdiv1  }
     DM(I2,M1) = B#0100000100011111;  { sport1 control
register: internal sclk & rfs, normal framing mode
frame sync not inverted 16-bit word length }
{====Initialize the DSP's interrupt registers==============}
     ICNTL=0x17;
     IMASK=B#000100;         {only SPORT1 tx interrupt
            enabled initially while in control mode }
{====EPROM version does not need the next two lines.
Sampling freq is set in set_control.  NOTE this is also the
bit rate for transmission===}
     AX0 = 0X2104;             {set sample freq 5.5125 kHz}
     DM(CTRLOUT) = AX0;
{.....Set bit test mask for DCB bit, used in tx interrupt
state machine....}
     AY0=DM(CTRLOUT); { test mask for DCB bit }
{..send first control word to switch codec to data mode ...}
     AX0=DM(I6,M7);      { send first 16bits of ctrl word }

     TX1=AX0;
     I2=0x3ffe;
```

```
     DM(I2,M1) = 0x0000;  {No Wait states}
     DM(I2,M1) = 0x0c18;   {system control reg: sport1
                    enabled }

{..... Wait for an interrupt indicating that transmit
register is ready for new data and that the 2105 has
received a 16bit word.....}

WAIT1:     AX1=DM(DMODE_FLG);   { check dmode flag }
     AR=PASS AX1;
     IF GT JUMP GO_DMODE; { if set, in data mode }
     JUMP WAIT1;          {  else, wait for initialization to
be completed from tx interrupt routine }

GO_DMODE:  L7=%DATAIN;      { init I7, L7 for rx autobuffer }
     I7=^DATAIN;
     L6=%DATAOUT_;        { init I6, L6 for tx autobuffer }
     I6=^DATAOUT_;
     AX0=DM(I6,M7);          { send first 16bits of data }
     TX1=AX0;
     AX0=0X0c18;
     DM(0X3FFF)=AX0;         { turn on sport1 }
     IFC=B#000000111111;    { clear all pending interrupts }
     nop;                    { cycle for IFC latency }
     ENA TIMER;
     IMASK=B#000011;    { sport1 rx and timer interrupt on }


main_loop:
    AX0 = DM(data_val_);
    AR=PASS AX0;
    IF EQ JUMP main_loop;
secnd_loop:
    AX0 = DM(flip);
    AR=PASS AX0;
    IF EQ JUMP NEXT;          {get freshest data}
    AX0=DM(DATAOUT_+1);       {check right channel against
                    threshold}
    AY1=threshold;
    AR=AX0-AY1;
    IF LT JUMP secnd_lo  ;
    AX0 = 1;
    DM(SIGNAL_FLAG) = AX0;
    JUMP secnd_loop;
NEXT:
    AX0=DM(DATAOUT_+5);{check right channel against
                    threshold}
    AY1=threshold;
    AR=AX0-AY1;
    IF LT JUMP secnd_loop;
    AX0 = 1;
```

```
    DM(SIGNAL_FLAG) = AX0;
    JUMP secnd_loop;

{=======Interrupt routines==================================}
{ Note: AY0 contains a bit mask and must NOT be modified
elsewhere          }
{..........................................................}
SETUPCONTROL:  AX0=DM(FIRST_FLG);   { first time through ? }

     AF=PASS AX0;
     IF NE JUMP DECR_FIRST;  { if so, wait until next word
                    transmitted}
     AX0=DM(DCB_FLG);
     AR=PASS AX0;
     IF EQ JUMP DCBFLG_SET;
     AX0=DM(CTRLIN);          {DCB_FLG has not been set yet}
     AR=AX0 XOR AY0;          {check all incoming bits
                    including DCB bit}
     IF EQ JUMP SET_DCB;      {set flag if DCB was 0}
     RTI;
DCBFLG_SET:  AX0=DM(CTRLIN);      {DCB_FLG was set}
     AR=AX0 AND AY0;          {only check for DCB bit}
     IF NE JUMP SETDMODE;     {if DBC=1 ready for datamode}
     RTI;
SET_DCB: AX0=0;
    DM(DCB_FLG)=AX0;
    AY0=0x0400;
    AX0=DM(CTRLOUT); {DCB was 0, prepare to send DCB=1,
                    DFR=0}
    AR = AX0 OR AY0;
    DM(CTRLOUT)=AR;
    RTI;

DECR_FIRST:  AX0=0;
     DM(FIRST_FLG)=AX0;      { if first time, set flag=0 }
     RTI;

SETDMODE:         IMASK=0;
         AX0=0X0418;               {disable sport1}
         DM(0X3FFF)=AX0;

         I6 = ^DATAOUT_;
         L6=0;
         DM(I6,M7) = 0x0000;  {reset output & input data
                    buffers}
         DM(I6,M7) = 0x0000;  {initialize embedded control
                    bits}
         DM(I6,M7) = 0xC000;  {out line 1&2 enab,0 out
                    atten,speaker mute}
         DM(I6,M7) = 0x40F0;       {PIO, etc}
```

```
            {To set the digital output pin (open-collector)
set PIO1 to the desirable value.}
            {PIO=11,OVR=0,IS=0,LG=0,MA=15,RG=0}
          DM(I6,M7) = 0x0000;     {reset output & input data
                        buffers}
          DM(I6,M7) = 0x0000;     {initialize embedded control
bits}
          DM(I6,M7) = 0xC000;        {same as aboveC000}
          DM(I6,M7) = 0x40F0;
{PIO=01,OVR=0,IS=0,LG=0,MA=15,RG=0}
            {To set the digital output pin (open-collector)
set PIO1 to the desirable value.}
          AX0=0X001F;
          DM(0X3FF2)=AX0;  { sport1 control: external tfs
                           external sclk & rfs
                           16 bit words }
          AX0=1;
          dm(mode_sel)=AX0;          { set D/C high }
          DM(DMODE_FLG)=AX0;         { set data mode flag
                                     high }
          RTI;

  NEWDATA:
    DIS BIT_REV;
    DM(AR_save) = AR;
    DM(AX0_save) = AX0;
    DM(AY1_save) = AY1;
    AY1 = DM(flipper);
    AX0 = txn_divisor;  {this number is used as a divisor to
                                  slow transmission rate}
    AR = AX0 - AY1;
    IF EQ CALL reset_flipper;  {reset flipper and assign or
                                     incr buffer}
    AY1 = DM(flipper);
    AR = AY1 + 1;
    DM(flipper) = AR;
    AX0 = DM(flip);
    AR = PASS AX0;
    IF NE JUMP second_half;

    AX0=DM(DATAIN);      { get LEFT channel data }
    DM(DATAOUT_)=AX0;     { output LEFT channel data }
    AX0=DM(DATAIN+1);    { get RIGHT channel data }
    DM(DATAOUT_+1)=AX0;  { output RIGHT channel data }
    AX0 = DM(cntrl_2);
    DM(DATAOUT_+2)=AX0;
    AX0 = DM(cntrl_1);
    DM(DATAOUT_+3)=AX0;
    AX0 = 1;             { Toggle "flip"}
    JUMP nd_end;
```

```
second_half:
    AX0=DM(DATAIN);      { get LEFT channel data }
    DM(DATAOUT_+4)=AX0;    { output LEFT channel data }
    AX0=DM(DATAIN+1);    { get RIGHT channel data }
    DM(DATAOUT_+5)=AX0;  { output RIGHT channel data }
    AX0 = DM(cntrl_2);
    DM(DATAOUT_+6)=AX0;
    AX0 = DM(cntrl_1);
    DM(DATAOUT_+7)=AX0;
    AX0 = 0;             { Toggle "flip"}

nd_end:
    DM(flip) = AX0;
    AY1=DM(num_samples); {the following code sets the
                    data_val_ flag}
    AX0=min_samples;
    AR = AY1-AX0;
    IF GT JUMP clean_up;
    IF EQ JUMP clean_up_1;
    AR = AY1 + 1;
    DM(num_samples) = AR;
    JUMP clean_up;
clean_up_1: AX0 = 1;
    DM(data_val_) = AX0;
clean_up:
    AY1 = DM(AY1_save);        {restore register}
    AX0 = DM(AX0_save);
    AR = DM(AR_save);
    RTI;                 { return }
 HOUR:
     DM(AX0H_save) = AX0;
     DM(AY1H_save) = AY1;       {need to store AR}
     AY1 = DM(NUM_SEC);
     AX0 = DM(signal_flag);
     AR = PASS AX0;
     IF GT JUMP hour_end;      {prevents overriding an alarm
                    condition}
     AR = PASS AY1;
     IF GT JUMP hour_end;       {only send every NUM_SEC
                    seconds}
     AY1 = one_hour;
     AX0 = -1;
     DM(signal_flag) = AX0; {set flag to indicate IAMOK}
 hour_end:
     AR = AY1 - 1;
     DM(NUM_SEC) = AR;
     AX0 = DM(AX0H_save);
     AY1 = DM(AY1H_save);            {need to restore AR}
 RTI;
.ENDMOD;
```

C-LINK.BAT

This file is a DOS batch file that assembles, links, and PROM splits two input files. One of these files (clink.c) is a C-language file that uses an infinite loop to continuously monitor sampled input to determine if an ALARM condition exists. If an ALARM condition does exist it sets SIGNAL_FLAG to the appropriate value. The other file (2105_hdr.dsp) is an assembly language file that contains the interrupt table, interrupt handlers, and signal array initializations. The header file (2105_hdr.dsp) is required for a C-program to operate. A detailed description of this batch file can be found in Appendix D.

```
asm21 2105_hdr.dsp -l -c -s
asm21 -l -c -s -cp -DDMSTACK -DIMAGE=RAM -DANY=RAM
frame_lg.dsp
g21 clink.c frame_lg.obj -a 2105.ach -mreserved=i2,i3 -v
-runhdr 2105_hdr.obj -g -save-temps -Wall -o cexample
spl21 cexample cprom -i -loader -bs 1024 -bb 2048
```

CLINK.C

This file is a C-language program that uses an infinite loop to continuously monitor sampled input to determine if an ALARM condition exists. If an ALARM condition does exist it sets SIGNAL_FLAG to the appropriate value.

```
extern int SIGNAL_FLAG;
extern int data_val;
extern int DATAOUT[];

void
main()
{
   int right1,right2,check_data,sat;
   sat=1;
loop1:
   check_data=data_val;  /*do nothing until the data is
                         valid*/
   if (!check_data) goto loop1;  /*"goto" used because
            optimizer prevents using do-while loop*/
loop2:
   right1=DATAOUT[1];
   right2=DATAOUT[5];
   if (right1>255 || right2>255) SIGNAL_FLAG=1;/* threshold
                         is 255*/
   goto loop2;
}
```

2105_HDR.DSP

This file is a modified version of transmit.dsp. It is loaded at absolute address 0x0000 and is responsible for establishing the interrupt table. In addition, for this program to link with a C-program, the only registers that could be changed permanently are I2 and I3. For this reason the autobuffering registers had to be changed to I2 and I3.

```
.MODULE/RAM/ABS=0x0000 ADSP2105_Runtime_Header; {NOTE: this
                       must be abs0000}
.CONST SYS_CTL_REG=0x3FFF; (system control register}
.CONST mode_sel=0x1000; {latched control for Control / Data
                       line }
{*************************************************************}
.VAR/RAM/CIRC  CTRLIN[4]; {circular buffers for data input
and}
.VAR/RAM/CIRC  CTRLOUT[4]; {output for data mode and control
                       mode}
.VAR/RAM/CIRC  DATAIN[4];
.VAR/RAM/CIRC  DATAOUT_[8];
.var/ram       ALARM[16];  {signal buffers should be 16 long
                       for DEDSEC}
.var/ram       IAMOK[16];  {all buffers must be the same
                       length}
.var/ram       NORMAL[16];
.var/ram       SIGNAL_FLAG_, LOOP_CNTR;
.VAR/RAM       FIRST_FLG; {First time thru flag}
.VAR/RAM       DCB_FLG; {DCB software handshaking flag
                       between 1849 & 2105}
.VAR/RAM       DMODE_FLG; {1849 mode flag i.e. CONTROL or
                       DATA modes}
.VAR/RAM       cntrl_1,cntrl_2, AR_save, AX0_save, flip,
            flipper, AY1_save;
.VAR/RAM       AX0H_save, AY1H_save, ARH_save, NUM_SEC,
            I0_save, buff_addr;
.VAR/RAM       L0_save, num_samples, data_val_;
.CONST one_hour=17;     {1:1.67 sec ratio @ 5 kHz sampling,
                       2160=1 hour}
.CONST txn_divisor=2000;  {used to slow transmission rate, 1
                       is minimum}
.CONST min_samples=300;  {this is the min num of samples to
                       ensure that the available data is
                       valid.}

.ENTRY      ___lib_prog_term;

.EXTERNAL   ___lib_setup_everything;
.EXTERNAL      main_;
```

```
.global     SIGNAL_FLAG_;
.global     DATAOUT_;
.global     data_val_;

{============= INTERRUPT VECTORS ========================}
code_start:  JUMP through;nop;nop;nop;
irq2_intr: RTI;nop;nop;nop;          {Sample clock interrupt}
sport_txm_intr: RTI;nop;nop;nop; {Serial port transmit
                    interrupt}
sport_rec_intr: RTI;nop;nop;nop; {Serial port receive
                    interrupt}
irq1_intr: JUMP SETUPCONTROL;nop;nop;nop;
irq0_intr: JUMP NEWDATA;nop;nop;nop;  {Receive interrupt }
timer_intr: JUMP HOUR;nop;nop;nop;    {Internal timer
                    interrupt}


set_control:
    AX1 = 0x2104;  {samp freq=5.125 khz,stereo,pcm16,2104}
    DM(CTRLOUT) = AX1;
    AX1 = 0x2200;    {xtal2,64bits/frame,master,serial txn}
    DM(CTRLOUT+1) = AX1;
    AX1 = 0xC0F0;    {(PIO=11, etc) Data mode parameter
                    defaults}
    DM(cntrl_1) = AX1;
    AX1 = 0xC000;   {OM, etc}
    DM(cntrl_2) = AX1;
    RTS;

reset_flipper:
    DM(I0_save)=I0;
    DM(L0_save)=L0;
    AX0 = 0;      {reset flipper and assign or step buffer}
    DM(flipper) = AX0;
    AY1 = DM(LOOP_CNTR);
    AR = AY1 +1;
    DM(LOOP_CNTR) = AR;
    AX0 = AR;
    AY1 = %IAMOK;   {could have used any msg buffer}
    AR = AX0 - AY1;
    IF LT JUMP FINISH1;
         AX0 = 0;
         DM(LOOP_CNTR) = AX0;
         AX0=DM(SIGNAL_FLAG_);
         AR=PASS AX0;
         IF EQ JUMP NORMAL_XMIT;
         IF GT JUMP ALARM_XMIT;
IAMOK_XMIT:  I0=^IAMOK;
         L0=0;
         AX0=0;          {return to normal after transmit}
         DM(SIGNAL_FLAG_)=AX0;
```

```
          JUMP FINISH2;

NORMAL_XMIT:    I0=^NORMAL;
          L0=0;
          JUMP FINISH2;

ALARM_XMIT:  I0=^ALARM;
          L0=0;
          AX0=0;          {return to normal after transmit}
          DM(SIGNAL_FLAG_)=AX0;
          JUMP FINISH2;

FINISH1: I0 = DM(buff_addr);
FINISH2: AX0 = DM(I0,M1);
                    DM(cntrl_1) = AX0;
     DM(buff_addr) = I0;
     I0 = DM(I0_save);
     L0 = DM(L0_save);
     RTS;


fill_mssg_buffer: {fill buffers for auto transmit}
{===MSB is the transmit bit (pio1).  MSB-1 is the data bit
(pio0). ========}
     I0=^ALARM;         {id=0x0253, alarm=lsb=1}
     L0=0;
     DM(I0,M1)=0X00F0;  {P0}
     DM(I0,M1)=0xC0F0;          {C0}
     DM(I0,M1)=0x00F0;          {C1}
     DM(I0,M1)=0xC0F0;          {D1}
     DM(I0,M1)=0x00F0;          {C2}
     DM(I0,M1)=0xC0F0;          {D2}
     DM(I0,M1)=0x00F0;          {D3}
     DM(I0,M1)=0xC0F0;          {D4}
     DM(I0,M1)=0x00F0;          {C3}
     DM(I0,M1)=0xC0F0;          {D5}
     DM(I0,M1)=0x00F0;          {D6}
     DM(I0,M1)=0xC0F0;          {D7}
     DM(I0,M1)=0x00F0;          {D8}
     DM(I0,M1)=0xC0F0;          {D9}
     DM(I0,M1)=0x00F0;          {D10}
     DM(I0,M1)=0xC0F0;          {ALARM BIT}

     I0=^IAMOK;
     L0=0;
     DM(I0,M1)=0X40F0;          {P0}
     DM(I0,M1)=0x80F0;          {C0}
     DM(I0,M1)=0x40F0;          {C1}
     DM(I0,M1)=0x80F0;          {D1}
     DM(I0,M1)=0x40F0;          {C2}
     DM(I0,M1)=0x80F0;          {D2}
```

```
    DM(I0,M1)=0x40F0;          {D3}
    DM(I0,M1)=0x80F0;          {D4}
    DM(I0,M1)=0x40F0;          {C3}
    DM(I0,M1)=0x80F0;          {D5}
    DM(I0,M1)=0x40F0;          {D6}
    DM(I0,M1)=0x80F0;          {D7}
    DM(I0,M1)=0x40F0;          {D8}
    DM(I0,M1)=0x80F0;          {D9}
    DM(I0,M1)=0x40F0;          {D10}
    DM(I0,M1)=0x80F0;          {ALARM BIT}

    I0=^NORMAL;
    L0=0;
    DM(I0,M1)=0x40F0;        {ensures transmitter is off}
    DM(I0,M1)=0x40F0;
    DM(I0,M1)=0x40F0;
    DM(I0,M1)=0x40F0;
    DM(I0,M1)=0x40F0;
    DM(I0,M1)=0x40F0;
    DM(I0,M1)=0x40F0;
    DM(I0,M1)=0x40F0;
    DM(I0,M1)=0x40F0;
    DM(I0,M1)=0x40F0;
    DM(I0,M1)=0x40F0;
    DM(I0,M1)=0x40F0;
    DM(I0,M1)=0x40F0;
    DM(I0,M1)=0x40F0;
    DM(I0,M1)=0x40F0;
    DM(I0,M1)=0x40F0;
    DM(I0,M1)=0x40F0;
    RTS;


through:
    CALL ___lib_setup_everything;
    IFC=H#003F;{CLEAR ALL PENDING INTERRUPTS}
    IMASK=H#0000;
    CALL set_control;
    AX0 = %IAMOK - 1; {could have used any buffer for
                       length}
    DM(LOOP_CNTR) = AX0;
    AX0 = txn_divisor;
    DM(flipper) = AX0;
    AX0 = 1;
    DM(flip) = AX0;
    AX0=0;
    DM(num_samples) = AX0;
    DM(data_val_) = AX0;
    DM(mode_sel) = AX0;
    DM(SIGNAL_FLAG_) = AX0;
    AX0 = 0xFFFF;
    DM(0x3FFD) = AX0;        {load TPERIOD}
```

```
    DM(0x3FFC) = AX0;       {load TCOUNT}
    AX0 = 0xFF;
    DM(0x3FFB) = AX0;        {load TSCALE}
    AX0 = one_hour;
    DM(NUM_SEC) = AX0;
{====Initialize the transmit buffers=======================}
    M1=1;
    CALL fill_mssg_buffer;
{====Initilaize the addressing registers of 2105===========}
    L3=%CTRLIN;
    I3=^CTRLIN;
    L2=%CTRLOUT;
    I2=^CTRLOUT;
{====Initialize software flags=============================}
    AX0=1;
    DM(FIRST_FLG)=AX0;
    DM(DCB_FLG)=AX0;
    AX0=0;
    DM(DMODE_FLG)=AX0;       { in control mode }
    AY0=DM(CTRLOUT);

{====Initialize the DSP's SPORT1 Serial port registers=====}

     L0 = 0;              { linear addressing for register }
     I0 = 0x3fef;           { point to last DM cntrl reg }
     DM(I0,M1) = 0x04B7;    { I2,M1,I3,M1 sport1 autobuffer
                    register }
     DM(I0,M1) = 383;          { rfsdiv1   }
     DM(I0,M1) = 849;          { sclkdiv1  }
     DM(I0,M1) = B#0100000100011111;  { sport1 control
register: internal sclk & rfs, normal framing mode
frame sync not inverted, 16-bit word length }
{====Initialize the DSP's interrupt registers==============}
     ICNTL=0x17;
     IMASK=B#000100;         {only SPORT1 tx interrupt
            enabled initially while in control mode }
{.....Set bit test mask for DCB bit, used in tx interrupt
state machine....}
     AY0=DM(CTRLOUT); { test mask for DCB bit }
{.send first control word to switch codec to data mode....}
     AX0=DM(I2,M1);     { send first 16bits of ctrl word }

     TX1=AX0;
     I0=0x3ffe;
     L0=0;
     DM(I0,M1) = 0x0000;  {No Wait states}
     DM(I0,M1) = 0x0c18;   {system control reg: sport1
                    enabled }
```

```
{..... Wait for an interrupt indicating that transmit
register is ready for new data and that the 2105 has
received a 16bit word.....}

WAIT1:     AX1=DM(DMODE_FLG);   { check dmode flag }
     AR=PASS AX1;
     IF GT JUMP GO_DMODE; { if set, in data mode }
     JUMP WAIT1;          {  else, wait for initialization to
be completed from tx interrupt routine }

GO_DMODE:
     L3=%DATAIN;           { init I3, L3 for rx autobuffer }
     I3=^DATAIN;
     L2=%DATAOUT_;         { init I2, L2 for tx autobuffer }
     I2=^DATAOUT_;
     AX0=DM(I2,M1);          { send first 16bits of data }
     TX1=AX0;
     AX0=0x0c18;
     DM(0x3FFF)=AX0;         { turn on sport1 }
     IFC=B#000000111111;    { clear all pending interrupts }
     nop;                    { cycle for IFC latency }
     ENA TIMER;
     IMASK=B#000011;    { sport1 rx and timer interrupt on }

     CALL main_;        {Begin C program}
___lib_prog_term: JUMP ___lib_prog_term;

{=======Interrupt routines=================================}
{ Note: AY0 contains a bit mask and must NOT be modified
elsewhere           }
{.........................................................}
SETUPCONTROL:  AX0=DM(FIRST_FLG);   { first time through ? }

     AF=PASS AX0;
     IF NE JUMP DECR_FIRST;  { if so, wait until next word
                               transmitted}
     AX0=DM(DCB_FLG);
     AR=PASS AX0;
     IF EQ JUMP DCBFLG_SET;
     AX0=DM(CTRLIN);          {DCB_FLG has not been set yet}
     AR=AX0 XOR AY0;          {check all incoming bits
    including DCB bit}
     IF EQ JUMP SET_DCB;      {set flag if DCB was 0}
     RTI;
DCBFLG_SET:  AX0=DM(CTRLIN);      {DCB_FLG was set}
     AR=AX0 AND AY0;          {only check for DCB bit}
     IF NE JUMP SETDMODE;     {if DBC=1 ready for datamode}
     RTI;
SET_DCB: AX0=0;
    DM(DCB_FLG)=AX0;
    AY0=0x0400;
```

```
    AX0=DM(CTRLOUT); {DCB was 0, prepare to send DCB=1,
                      DFR=0}
    AR = AX0 OR AY0;
    DM(CTRLOUT)=AR;
    RTI;

DECR_FIRST:  AX0=0;
     DM(FIRST_FLG)=AX0;      { if first time, set flag=0 }
     RTI;

SETDMODE:         IMASK=0;
         AX0=0x0418;                {disable sport1}
         DM(0x3FFF)=AX0;

         I2 = ^DATAOUT_;
         L2=0;
         DM(I2,M1) = 0x0000;        {reset output & input
                                     data buffers}
         DM(I2,M1) = 0x0000;        {initialize embedded
                    control bits}
         DM(I2,M1) = 0xC000;        {out line 1&2 enab,0 out
                                     atten,speaker mute}
         DM(I2,M1) = 0x40F0;        {PIO, etc}
{To set the digital output pin (open-collector) set PIO1 to
the desirable value.}
                     {PIO=11,OVR=0,IS=0,LG=0,MA=15,RG=0}
         DM(I2,M1) = 0x0000;        {reset output & input
                    data buffers}
         DM(I2,M1) = 0x0000;        {initialize embedded
                    control bits}
         DM(I2,M1) = 0xC000;        {same as aboveC000}
         DM(I2,M1) = 0x40F0;
{PIO=01,OVR=0,IS=0,LG=0,MA=15,RG=0}
{To set the digital output pin (open-collector)set PIO1 to
the desirable value.}
         AX0=0x001F;
         DM(0x3FF2)=AX0;     { sport1 control: external tfs,
                           external sclk & rfs 16 bit words }
         AX0=1;
         dm(mode_sel)=AX0;          { set D/C high }
         DM(DMODE_FLG)=AX0;     { set data mode flag high }
         RTI;


 NEWDATA:
    DIS BIT_REV;
    DM(AR_save) = AR;
    DM(AX0_save) = AX0;
    DM(AY1_save) = AY1;
    AY1 = DM(flipper);
```

```
    AX0 = txn_divisor;    {this number is used as a divisor
                      to slow transmission rate}
    AR = AX0 - AY1;
    IF EQ CALL reset_flipper;  {reset flipper and assign or
                      incr buffer}
    AY1 = DM(flipper);
    AR = AY1 + 1;
    DM(flipper) = AR;
    AX0 = DM(flip);
    AR = PASS AX0;
    IF NE JUMP second_half;

    AX0=DM(DATAIN);      { get LEFT channel data }
    DM(DATAOUT_)=AX0;     { output LEFT channel data }
    AX0=DM(DATAIN+1);    { get RIGHT channel data }
    DM(DATAOUT_+1)=AX0;  { output RIGHT channel data }
    AX0 = DM(cntrl_2);
    DM(DATAOUT_+2)=AX0;
    AX0 = DM(cntrl_1);
    DM(DATAOUT_+3)=AX0;
    AX0 = 1;             { Toggle "flip"}
    JUMP nd_end;

second_half:
    AX0=DM(DATAIN);      { get LEFT channel data }
    DM(DATAOUT_+4)=AX0;    { output LEFT channel data }
    AX0=DM(DATAIN+1);    { get RIGHT channel data }
    DM(DATAOUT_+5)=AX0;  { output RIGHT channel data }
    AX0 = DM(cntrl_2);
    DM(DATAOUT_+6)=AX0;
    AX0 = DM(cntrl_1);
    DM(DATAOUT_+7)=AX0;
    AX0 = 0;             { Toggle "flip"}

nd_end:
    DM(flip) = AX0;
    AY1=DM(num_samples); {the following code sets the
                      data_val_ flag}
    AX0=min_samples;
    AR = AY1-AX0;
    IF GT JUMP clean_up;
    IF EQ JUMP clean_up_1;
    AR = AY1 + 1;
    DM(num_samples) = AR;
    JUMP clean_up;
clean_up_1: AX0 = 1;
    DM(data_val_) = AX0;
clean_up:
    AY1 = DM(AY1_save);    {restore registers}
    AX0 = DM(AX0_save);
    AR = DM(AR_save);
```

```
    RTI;                    { return }


 HOUR:
      DM(AX0H_save) = AX0;
      DM(AY1H_save) = AY1;
      DM(ARH_save) = AR;
      AY1 = DM(NUM_SEC);
      AX0 = DM(SIGNAL_FLAG_);
      AR = PASS AX0;
      IF GT JUMP hour_end;  {prevents overriding an alarm
condition}
      AR = PASS AY1;
      IF GT JUMP hour_end;       {only send every NUM_SEC
                        seconds}
      AY1 = one_hour;
      AX0 = -1;
      DM(SIGNAL_FLAG_) = AX0; {set flag to indicate IAMOK}
 hour_end:
      AR = AY1 - 1;
      DM(NUM_SEC) = AR;
      AX0 = DM(AX0H_save);
      AY1 = DM(AY1H_save);
      AR = DM(ARH_save);
 RTI;

.ENDMOD;
```

APPENDIX D: PROCEDURES FOR COMPILING NEW CODE AND EPROM LOADING

Compiling New Code:

Below is a listing of the DOS batch file C-LINK.BAT (also listed in Appendix C). This file causes several actions to occur.

```
asm21 2105_hdr.dsp -l -c -s
asm21 -l -c -s -cp -DDMSTACK -DIMAGE=RAM -DANY=RAM
frame_1g.dsp
g21 clink.c frame_1g.obj -a 2105.ach -mreserved=i2,i3 -v
-runhdr 2105_hdr.obj -g -save-temps -Wall -o cexample
spl21 cexample cprom -i -loader -bs 2048 -bb 2048
```

The first line assembles the header file 2105_hdr.dsp. This file contains the interrupt table settings, interrupt handlers and initialization routines.

The second line was included to correct a bug in the software sent by Analog Devices. The bug was in the file frame_1g.dsp. This program is used by C-routines when calling and returning from subroutines. It is only supposed to push and pop non-dag (non-Data Address Generator) registers (M,I,and L registers) on a stack. The bug came from the fact that the routine would push and pop registers I2 and I3. These registers are supposed to be reserved for the user and are the only registers available for autobuffering. Without the modification to frame_1g.dsp, a return from some routine could possibly load an erroneous value into I2 and I3. This would cause unpredictable results in operations using autobuffering. The file frame_1g.dsp can be corrected by removing all operations that affect I2 and I3.

The third command line compiles clink.c and links it with frame_1g.obj and 2105_hdr.obj. If more than one file C-file is to be compiled and linked, simply replace clink.c with *@files_all*. Where *files_all* is an ASCII file containing the C-files to be compiled (one path\filename per line).

The last line is the PROM splitter command line. The PROM slitter converts executable into a format that can be downloaded onto an PROM or EPROM. The -i switch converts the executable code into Intel hex format. The -loader switch automatically splits large code up between the boot pages as specified by the -bs and the -bb switches. The -bs switch specifies the boot page size in while the -bb switch specifies the boot boundaries. The splitter will put the output in a file named cprom.bnm.

EPROM Loading

The following procedure describes the process for programming an EPROM at the Naval Postgraduate School.

1. Erase one D27C512 EPROM for 20-30 minutes under an UV light. The digital lab manager in Bullard has an UV light available for this purpose.

2. Ensure the cprom.bnm file is loaded on a 360KB 5.25" floppy disk and insert disk containing *.bnm in drive a:\.

3. While the EPROM is being erased, goto the PC in the digital lab that has the Modular Circuit Technology's (MCT) PROM burner installed and type *mct* at the C:> prompt.

4. At the main menu select the *Utility* option.

5. Pick opt. *0* to convert the *.bnm file to binary format.

6. Enter the full path name of the input file when prompted (i.e. a:\cprom.bnm).

7. Enter the output file name when prompted.

8. Enter *I* (Intel) for source file hex format.

9. Choose the *Programmer* option from the main menu.

10. Choose option *1* at the next menu to program an EPROM.

11. At the *MFR* prompt enter *08* for Intel chip.

12. Enter *12* for the type of chip (D27C521)

13. At the next menu choose option *2* to load the binary file into the buffer.

14. At filename prompt, enter the full path name of the binary file (i.e. a:\cprom.bin).

15. Set load address = *0*.

16. Place an erased EPROM in the MCT unit as illustrated on top of the unit.

17. Select option *A*. This will blank check, program, and verify the EPROM.

18. If the erased EPROM is locked into the MCT ZIF socket, enter *Y* at the "Ready to Start?" prompt.

19. When the verification is complete enter <CR> at the prompt and remove the EPROM.

20. With power secured to the DSP board, insert the newly programmed EPROM into Boot Memory socket. Ensure chip orientation is correct. Pin #1 should be closest to jumpers J1 and J2.

21. Program execution will begin at power-up or when reset switch S1 is pressed.

LIST OF REFERENCES

1. Department of Defense, Unmanned Air Vehicles (UAV) Master Plan, Program Executive Officer, Mr. Robert Glomb, Washington D.C., March 1993.

2. Kaltenberger, B. R., *Unmanned Air Vehicle/Remotely Piloted Vehicle Analysis for Lethal UAV/RPV*, Master's Thesis, Naval Postgraduate School, Monterey, California, September 1993.

3. Dickson,Paul, "Wiring down the war," in *The Electronic Battlefield*, Indiana University Press, Bloomington, Indiana, 1976, pp. 96-98.

4. Rodgers, A. L., and others, *Surveillance and Target Acquisition Systems*, Brassey's Defence Publishers, Oxford, 1983, pp. 164-170.

5. Bergin J. D., "The electronic battlefield," in *United States Army in Vietnam. Military Communications, a Test for Technology*, Center of Military History, United States Army, Washington, D.C., 1986, pp. 392-393.

6. *Jane's Battlefield Surveillance Systems*, E. R. Hooton, and Kenneth Munson, eds., 5th ed., Jane's Information Group Limited, Coulsdon, Surrey, United Kingdom, 1993, pp. 51-52.

7. Interview between Mr. T. Reynolds, Mr. W. Dense, Mr. P. Winters, Naval Surface Warfare Center, Silver Springs, Maryland, and the author, 19 November 1993.

8. Stremler, F. G., *Introduction to Communication Systems*, Third Edition, Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1992, pp.589-594.

9. Innovative Integration, *SBC-31 Hardware Reference Manual*, Innovative Integration, Inc., Moorpark, California, 1994, pp. 1-11.

10. *EMB-1601A Embedded Digital Signal Processor & DSP Development System*, Wavetron Microsystems, version 1.10, June 1993.

11. Telephone conversation between Brent Roman, Wavetron Microsystems and the author, 22 April 1994.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2.	Library, Code 52 Naval Postgraduate School Monterey, California 93943-5101	2
3.	Chairman, Code EC Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, California 93943-5121	1
4.	Professor Michael K. Shields, Code EC/SL Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, California 93943-5121	2
5.	Professor Murali Tummala, Code EC/TU Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, California 93943-5121	2
6.	Professor Richard M. Howard, Code AA/HO Department of Aeronautics and Astronautics Naval Postgraduate School Monterey, California 93943-5106	1
7.	Professor Isaac I. Kaminer, Code AA/KA Department of Aeronautics and Astronautics Naval Postgraduate School Monterey, California 93943-5106	1
8.	LT Donald B. Howard 3216 Marshall Rd. Ottawa, Kansas 66067	2
9.	Farid Dibachi Wavetron Microsystems 1197 Oddstad Drive Redwood City, California 94063	1